System-Level Synthesis of Ultra Low-Power Wireless Sensor Network Node Controllers: A Complete Design-Flow

Muhammad Adeel Ahmed Pasha

N° d'ordre : 4297 **ANNÉE 2010**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

Ecole doctorale : Matisse

présentée par

Muhammad Adeel Ahmed PASHA

Préparée à l'unité de recherche : **CAIRN/IRISA-UMR 6074**
Nom développé de l'unité **: Institut de Recherche en Informatique et Systèmes Aléatoires de Rennes**
Composante universitaire : **S.P.M.**

Synthèse de haut-niveau de contrôleurs ultra-faible consommation pour des réseaux de capteurs: un flot de conception complet

System-Level Synthesis of Ultra Low-Power Wireless Sensor Network Node Controllers: A Complete Design-Flow

Thèse soutenue à l'IRISA Rennes
le 15 décembre 2010
devant le jury composé de:

Tanguy RISSET
Président
Professeur, INSA Lyon, Lyon

Cécile BELLEUDY
Examinateur
Maître de Conférences, Université de Nice, Sophia Antipolis

Christian PIGUET
Rapporteur
Professeur, EPFL, Lausanne, Suisse

Frédéric PETROT
Rapporteur
Professeur, ENSIMAG, Grenoble

Olivier SENTIEYS
Directeur
Professeur, Université de Rennes 1, Lannion

Steven DERRIEN
Co-directeur
Maitre de Conférences, Université de Rennes 1, Rennes

Acknowledgments

First of all, countless thanks to Almighty Allah, Who has given me knowledge and courage to carry out this work. Secondly, I would like the readers to keep all the below-mentioned people in mind while reading this dissertation, without them this success would not have been possible.

I would like to begin by thanking my Ph.D. advisors, Olivier Sentieys and Steven Derrien, for their guidance, understanding, patience, always believing in my capabilities and most importantly, their friendship during my Ph.D. at IRISA Rennes. They provided me different flavors of care as they were my bosses, advisors, friends and brothers from time to time. They have always helped me to not only grow as a researcher and developer but also an independent thinker and a better human being. It is my honour to have worked with them.

I would also like to thank all my colleagues of CAIRN research team and administration staff at IRISA for creating such a pleasant work environment and for being there for me. In particular many thanks to Kevin Martin, Antoine Floch, Erwan Raffin, Laurent Perraudeau, Antoine Morvan, Naeem Abbas, Amit Kumar, Jeremie Guidoux, Loic Cloatre, Florent Berthelot, Georges Adouko, Maxime Naullet, Charles Wagner, François Charot, Christophe Wolinski, Patrice Quinton and Ludovic L'Hours with whom I worked closely and had many fruitful discussions. Many thanks to my colleagues at Lannion as well including Olivier Berder, Daniel Menard, Daniel Chillet, Sebastian Pillement, Emmanuel Casseau, Philippe Quemerais, Thomas Anger, Arnaud Carer, Vivek T.D., Shafqat Khan and others. I would also like to pay special thanks to CAIRN team-assistants Celine, Elise and Nadia for their support.

Most importantly I would like to thank my parents, Abdul Hameed and Latifan Bibi, for their faith in me, encouraging me to be as ambitious as I wanted and supporting me in all my endeavors. It was under their watchful eye that I gained so much drive and ability to tackle challenges head on. A very special thanks goes to my siblings, nephews and nieces as well for being there for me all the time and cheering me up.

I thank all my friends in France and in Pakistan for motivating me and regularly wishing me good luck. I am very grateful to all of them who made me feel at home, cared for and allowed me to worry only about my studies during my stay in France. I would also like to thank my teachers at Centre d'Etude de Langue of Colmar and at University of Nice Sophia-Antipolis.

Finally I would like to express my indebtedness to all the jury members for spending their precious time to read and accept my work.

This dissertation is dedicated to the light of useful knowledge that enlightens our lives.

Résumé

Les réseaux de capteurs sont une technologie dont l'évolution est très rapide et avec un grand nombre d'applications potentielles dans des domaines variés (e.g. en médecine, en surveillance de l'environnement ou de structures, ou encore en contexte militaire). La conception d'une plateforme matérielle pour un nœud de capteur est un véritable défi car elle est soumise à des contraintes sévères. Par exemple, comme les nœuds doivent être de taille et de coût limités, il doivent comporter une capacité limitée d'énergie et ils s'appuient donc sur des sources d'énergie non rechargeables (e.g. piles) ou récupérées dans l'environnement (e.g. cellules photovoltaïques). Comme le réseau doit de plus pouvoir fonctionner sans intervention pendant une très longue durée (des mois voire des années), la consommation d'énergie est souvent considérée comme la contrainte la plus forte. De nos jours, ces dispositifs s'appuient principalement sur des microcontrôleurs à très faible consommation disponibles commercialement. Ces processeurs offrent des puissances de calcul raisonnable pour des coûts et une consommation limités. Cependant, ils ne sont pas nécessairement complètement adaptés au contexte des réseaux de capteurs car basés sur une structure de calcul monolithique et généraliste.

Dans cette thèse, nous proposons un flot de conception depuis le niveau système pour une approche alternative et originale se basant sur le concept de micro-tâches matérielles avec coupure de la tension d'alimentation *(power gating)*. Dans cette approche les parties calcul et contrôle d'un nœud de capteur sont constituées d'un ensemble de micro-tâches matérielles qui sont activées selon un principe événementiel, chacune étant dédiée à une tâche spécifique du système, telle que le relevé de paramètres, la couche MAC, le routage ou le traitement des données. En combinant la spécialisation du matériel avec la coupure d'alimentation, nous réduisons de façon significative les énergies dynamique et statique d'un dispositif. Suivant la philosophie de nombreux environnements logiciels de programmation des réseaux de capteurs, notre flot de conception utilise l'association d'un langage spécifique (DSL pour *Domain Specific Language*) pour les spécifications système (interactions entre micro-tâches, gestion des événements et des ressources partagées, etc.) et de C-ANSI pour spécifier le comportement de chaque micro-tâche. Partant de ces spécifications et en utilisant des approches MDE *(Model Driven Engineering)* et des techniques de compilation reciblables, notre flot génère une description VHDL synthétisable de l'ensemble du sous-système de contrôle et de calcul d'un nœud de capteur.

Dans un but de validation expérimentale de l'approche, nous accomplissons tout d'abord des simulations au niveau transistor à l'aide de SPICE pour étudier les performances des coupures d'alimentation dans notre système. Ces coupures dynamiques au cours de l'exécution sont possibles avec des temps de commutation très faibles, de l'ordre de la centaine de nano-secondes. Ceci améliore les temps de réveil et de mise en veille d'au moins 50% en comparaison avec les microcontrôleurs à faible consommation. Un cas d'étude sur une application en réseau de capteurs a été spécifié puis synthétisé à l'aide de notre flot de conception matériel en un ensemble de micro-tâches associées à un moniteur système. Les résultats montrent que des gains en énergie dynamique d'un à deux ordres de grandeur sont possibles en comparaison avec des implémentations à base de microcontrôleurs comme le MSP430. De la même façon, des gains d'un ordre de grandeur en énergie statique sont également obtenus grâce à la réduction de l'espace mémoire et à l'utilisation de la coupure des tensions d'alimentation.

Abstract

Wireless Sensor Networks (WSN) are a fast evolving technology having a number of potential applications in various domains of daily-life, such as structural and environmental monitoring, medicine, military surveillance, robotic explorations etc. Engineering a WSN node hardware platform is known to be a tough challenge, as the design must enforce many severe constraints. For example, since WSN nodes must have small form-factors and limited production cost, it is not possible to provide them with large energy sources. In most cases they must rely on non-replenishing (e.g. battery) or self-sufficient (e.g. solar cells) sources of energy. As WSN nodes may have to work unattended for long durations (months if not years), their energy consumption is often the most critical design parameter. Moreover, as a WSN node remains idle during most of its life-time (with a duty cycle of less than 1%), special measures have to be taken to avoid the high static energy dissipation. WSN node devices have until now been designed using off-the-shelf low-power microcontroller units (MCUs) (such as the MSP430, the ARM-Cortex-M0 or the ATMega128L). These MCUs provide a reasonable processing power with low power consumption at an affordable cost. However, they are not necessarily well-suited for WSN node design as they are based on a general purpose compute engine.

In this thesis, we propose a complete system-level design-flow for an alternative approach based on the concept of power-gated hardware micro-tasks. In this approach, computation and control part of a WSN node is made of several hardware micro-tasks that are activated on an event-driven basis, each of them being dedicated to a specific task of the system (such as event-sensing, low-power MAC, routing, and data processing etc.). By combining hardware specialization with power-gating, we can drastically reduce both dynamic and static energy of the WSN node. Following the philosophy of many WSN software frameworks, our design-flow uses a combination of a textual Domain Specific Language (DSL) for system-level specifications (interactions between micro-tasks, event management, shared resources management etc.) and ANSI-C for specifying the behavior of each micro-task. Starting from these specifications and by using Model Driven Engineering (MDE) and retargetable compilation techniques, we are able to generate a synthesizable VHDL description of the whole computation and control subsystem of a WSN node. This VHDL description provides a direct path to ASIC/FPGA implementation.

For experimental validation of the approach, first of all, we performed SPICE transistor-level simulations to study the feasibility of using power-gating in our system. We found that the power-gating scheme happens to have very short switching-time delays, in the orders of a few hundred of nano-seconds. This improves the wake-up response time by at least 50% when compared to low-power MCUs such as the MSP430. A case-study example of a WSN application was conceived and by using our design-flow, VHDL codes for different hardware micro-tasks and system monitor were obtained. The synthesis results show that dynamic power savings by one to two orders of magnitude are possible w.r.t. MCU-based implementations. Similarly, static power savings of one order of magnitude are also obtained due to the reduction in data memory size and power-gating.

Contents

Chapter 0

Résumé étendu

0.1 Réseaux de capteurs sans fil

Les réseaux de capteurs sont une technologie dont l'évolution est très rapide et avec un grand nombre d'applications potentielles dans des domaines variés de notre vie quotidienne, e.g. en médecine, en surveillance de l'environnement ou de structures, en robotique ou encore en contexte militaire. Les avancées dans les technologies de l'électronique numérique, de la microélectronique ou de la micromécanique MEMS *(Micro-Electro-Mechanical-Systems)* ont facilité le développement de nœuds de capteur à faible coût, faible encombrement et faible consommation qui communiquent sans-fil de façon efficace sur de faibles distances. Par conséquent, le domaine émergent des réseaux de capteurs combine la mesure, le calcul et la communication de données dans un unique et minuscule nœud dit "de capteur". Ces systèmes contenant des milliers voire des dizaines de milliers de tels nœuds sont anticipés afin de révolutionner la façon dont l'humain travaille et vit.

Le principal challenge dans le domaine des réseaux de capteurs est de faire face aux difficultés liées aux contraintes sévères de ressources et d'énergie liées au nœud de capteur. Les processeurs contrôlant le nœud ne contiennent que quelques kilo-octets de mémoire et doivent cependant implémenter des protocoles réseaux complexes. Plusieurs contraintes émergent du fait que ces composants seront produits en grand nombre et doivent être de petite taille et peu chers. Chaque nouvelle génération de technologie silicium amène un plus grand nombre de transistors sur une même surface et résulte donc en deux scénarios distincts: (i) plus de fonctionnalités peuvent être placées sur un composant à surface constante ou (ii) la taille du composant et sa consommation peuvent être réduites pour la même fonctionnalité.

La plus forte des contraintes et aussi la plus complexe à respecter dans ce domaine des réseaux de capteurs est celle de la consommation d'énergie ou de puissance. La faible taille d'un nœud de capteur et ses besoins en autonomie limitent de façon très forte la réserve d'énergie disponible sur un de ces dispositifs. Ceci induit des limitations en termes de puissance de calcul et de mémoire disponibles et conduit à des problématiques représentant un véritable challenge en termes d'architectures. De nombreux dispositifs,

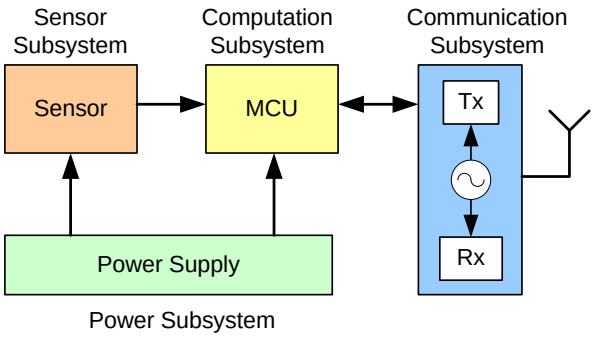


Figure 1: Architecture générale d'un nœud de capteur.

tels que les téléphones cellulaires et autres "*smartphones*", réduisent leur consommation d'énergie en utilisant des composants matériels spécialisés – ou *Application Specific Integrated Circuits (ASIC)* – qui procurent des implémentations faible consommation des protocoles de communications et algorithmes de traitement nécessaires.

A ce jour, une des forces des réseaux de capteurs est leur supposées flexibilité et universalité. Cependant, si on observe de façon plus précise leur conception actuelle, le besoin de flexibilité et de programmabilité est essentiellement poussé vers les couches hautes et applicatives, ce qui de plus ne représente qu'une faible fraction de la charge de calcul du nœud, alors que la plus grosse partie de celle-ci est dédiée à la pile de protocole de communication, en particulier vers les couches basses. Notre opinion est donc qu'il est intéressant d'explorer une approche de spécialisation du matériel dans la conception d'un nœud du réseau, de façon à respecter les exigences de très faible énergie. Afin de réduire cette consommation dans un dispositif, nous devons tout d'abord étudier son architecture générique et rechercher ses "points chauds". L'architecture générique d'un nœud de capteur est discutée dans la prochaine section.

0.1.1 Architecture d'un nœud de capteur

Les nœuds de capteur sont des dispositifs à faible consommation fortement embarqués constitués de blocs de calcul et de mémorisation (e.g. un microcontrôleur (MCU) connecté à une mémoire RAM et/ou flash) associés à des composants de communication sans-fil *(RF transceiver)* et à des capteurs/actionneurs. Comme les nœuds doivent être de taille et de coût limités, il doivent comporter une capacité limitée d'énergie [138]. Dans la plupart des cas, ils s'appuient donc sur des sources d'énergie non rechargeables (e.g. piles) ou récupérées dans l'environnement (e.g. cellules photovoltaïques).

La figure 1 présente l'architecture d'un nœud de capteur générique. Il est constitué de quatre sous-systèmes: alimentation, communication, contrôle et calcul, et capteurs. Le sous-système d'alimentation est constitué d'une batterie (ou d'une pile) et d'un convertisseur DC-DC. Le sous-système de communication constitué d'un émetteur-récepteur radio pour les communications sans-fil entre objets. La plupart des plate-formes utilise une antenne unique omnidirectionnelle, cependant des techniques de

coopération MIMO *(Multiple-Input and Multiple-Output)* peuvent également être déployées [99]. Le sous-système de calcul est typiquement composé de mémoire permettant de stocker le programme ou les données, et d'un microcontrôleur pour contrôler le système et traiter les données. Le dernier sous-système lie le nœud avec le monde physique et dispose d'un ensemble de capteurs et/ou d'actionneurs dépendant de l'application considérée. Il contient également des convertisseurs analogique-numérique pour convertir les signaux captés en données numériques utilisables pas le calculateur. Pour concevoir de tels dispositifs avec des ressources fortement limitées, les concepteurs d'architecture doivent faire face à des contraintes difficiles qui seront discutées dans la prochaine section.

0.1.2 Contraintes de conception d'un nœud de capteur

Concevoir ces nœuds de capteur est un réel challenge, car plusieurs fortes contraintes sont imposées qui, de plus, sont souvent étroitement liées. Les principales métriques utiles à cette conception sont décrites ci dessous.

- La ***consommation d'énergie*** (ou de puissance) est le plus grand challenge à atteindre car le réseau doit pouvoir fonctionner sans intervention pendant une très longue durée (des mois voire des années).
- La ***robustesse*** est également un critère important pour les réseaux de capteurs car elle permet de garantir le fonctionnement correct du r´eseau dans son ensemble. Chaque nœud doit donc être conçu pour être le plus robuste possible afin de tolérer et donc de s'adapter à des pannes de nœuds voisins.
- La ***sécurité*** au niveau de l'application est une autre métrique à considérer et les dispositifs doivent souvent embarquer des algorithmes relativement complexes d'authentification ou de chiffrement des données.
- Les ***débits et portées de communication*** sont des éléments clés dans la conception des nœuds. Augmenter la portée et le débit a cependant un impact significatif sur la consommation de puissance des partie radio et calcul.
- La ***charge de calcul*** est une autre métrique clé qui influence directement la consommation d'énergie du dispositif. Cependant, augmenter la puissance de calcul peut aussi permettre de réduire l'énergie du sous-système radio.
- Le ***coût et la taille*** de chaque nœud a un impact direct et significatif sur la facilité et le coût du déploiement du réseau de capteurs complet ainsi que sur la capacité de la source d'énergie disponible sur les dispositifs.

Comme discuté précédemment, les réseaux de capteurs sont déployés en général en grand nombre et ils doivent donc être petits et peu coûteux. Dans le même temps, comme il n'est pas possible de les équiper avec de grande sources d'énergie voire d'accéder à un rechargement de cette source, la très faible consommation est donc

leur contrainte majeure de conception. De plus, sachant que les nœuds sont inactifs ou en attente pendant la majeure partie de leur durée de vie (rapport cyclique de fonctionnement inférieur à 1%), l'énergie induite par les pertes statiques des composants (courants de fuite) doit être plus particulièrement réduite.

Si le profil de consommation d'un nœud de capteur est analysé sur l'ensemble de ses sous-systèmes, les blocs de communication et de calcul représentent bien sur la majeure partie du budget énergétique [116, 30]. Dans ce travail de thèse, nous ciblons par conséquent l'optimisation de la consommation des sous-systèmes de contrôle et de calcul. En effet, nous soutenons que les gains en énergie ou en puissance obtenus par notre approche sur ces blocs ouvrent des possibilités vers des protocoles de communication ou schémas de modulation plus complexes et plus coûteux en termes de puissance de calcul, ce qui, au final, permettra de fournir une meilleure qualité de service, une puissance radio plus faible et une meilleure efficacité d'utilisation de la bande passante du réseau. La réduction de puissance des sous-systèmes de contrôle et de calcul peut bénéficier d'optimisations à différents niveaux de conception du dispositif, tels que le niveau application, la micro-architecture ou le niveau de conception de circuits VLSI *(Very Large Scale Integrated)*.

Dans les paragraphes suivants, nous présentons et discutons les techniques de réduction de la consommation adaptées au domaine des réseaux de capteurs i.e. micro-architecture et conception VLSI.

0.2 Optimisation de puissance d'un nœud de capteur

Depuis un peu plus d'une décennie, les problèmes d'estimation de réduction de la consommation électrique dans les circuits VLSI sont l'objet d'un grand nombre de travaux de recherche. Une grand partie de ces travaux se sont notamment intéressés à la conception de microarchitectures faible consommation pour microcontrôleurs embarqués. Cette section a pour objectif d'offrir au lecteur une synthèse des techniques utilisées dans ce domaine.

0.2.1 Conception VLSI orientée faible consommation

La consommation électrique dans un circuit VLSI découle de deux phénomènes: la puissance dynamique, provoquée par les charges et décharges des capacités (découlant des changements d'états du circuit) et la puissance statique causée par les courants de fuites entre la tension d'alimentation et la masse.

Quand un circuit est en mode actif, la puissance qu'il dissipe est très largement dominée par sa composante dynamique, et peut-être considérée comme étant approximativement proportionnelle à sa fréquence d'horloge. Pour les réseaux de capteurs, les choses sont assez différentes car les nœuds peuvent rester inactifs pendant de longue périodes (taux d'activité souvent inférieur à 1%). Dans ce contexte, il n'est plus raisonnable d'ignorer la contribution de la puissance statique dans le bilan énergétique global.

Il existe de nombreuses techniques permettant de réduire la puissance dynamique dans un circuit (*clock gating*, contrôle de la tension d'alimentation, etc.), celles-ci pou-

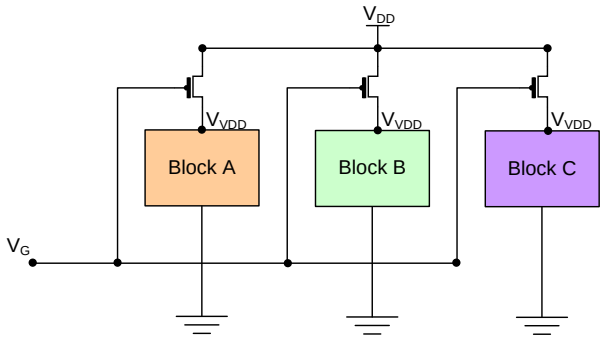


Figure 2: Un exemple d'utilisation du *power gating*.

vant être appliquées à différents étapes du flot de conception. La plupart d'entre elles sont cependant peu adaptées au contexte des réseaux de capteurs car elles ont souvent pour effet secondaire d'augmenter le nombre de transistors du circuit, augmentant ainsi indirectement sa puissance statique, avec un bilan global pouvant de fait devenir négatif.

La technique de la coupure des alimentations – ou *power gating* –, dont le principe consiste à couper l'alimentation d'un composant inactif [12, 87], est toutefois une exception à cette règle. Le *power gating*, si utilisé à bon escient, permet donc de réduire de fait à la fois la puissance dynamique et la puissance statique, ce qui en fait une technique particulièrement attrayante pour des circuits dont les périodes d'activité sont limitées.

La technique consiste à ajouter un *sleep transistor* entre la source d'alimentation V_{DD} globale et celle du composant, créant ainsi une *alimentation virtuelle* notée V_{VDD}, comme illustré sur la figure 2. Ce *sleep transistor*, lorsqu'il est ouvert, permet de réduire les courants de fuites du composant à leurs niveaux minimums

0.2.2 Microcontrôleurs faible consommation

Les microcontrôleurs à très faible consommation actuellement disponibles sur le marché (e.g. MSP430, CoolRISC and ATmega128L) partagent de très nombreuses caractéristiques : un chemin de données simple (8/16-bits), un faible nombre d'instructions (seulement 27 instructions pour le MSP430), et surtout de nombreux modes de fonctionnement qui permettent d'adapter dynamiquement le comportement du processeur en jouant sur des compromis entre gain en consommation et réactivité. Ces processeurs sont conçus pour une gamme d'application assez large et ne sont donc pas spécifiquement conçus pour des réseaux de capteurs. De fait, parce qu'il sont conçus sur la base d'un micro-architecture généraliste et monolithique, ils ne sont pas forcement bien adaptés à la nature très particulière (basée événements) de la charge de calcul de ces nœuds.

La plupart des plateformes matérielles utilisées dans des infrastructures de réseaux de capteurs utilisent des processeurs commerciaux de ce type. Par exemple, la plate-

forme Mica2 [25], qui a été très largement utilisée par la communauté, est basée sur un microcontrôleur ATmega128L de la société Atmel. Le même contrôleur a également été utilisé par les concepteurs de la plateforme *eXtreme Scale Mote* (XSM) [30]. Les autres plateformes (Hydrowatch [39], PowWow [64]) utilisent quand à elles des processeurs MSP430 [129] de la société Texas Instruments, tandis que la plateforme WiseNet est basée sur un processeur CoolRISC [33] de la société EM Microelectronic.

Bien que les niveaux de puissance dynamique relevés (et plus particulièrement en Joules/instruction) pour ces processeurs puissent sembler extrêmement faibles au regard de processeurs embarqués plus classiques (ex: MSP430), ces gains nous semblent cependant loin de ce qui pourrait être obtenu en combinant des approches exploitant la spécialisation et le parallélisme.

Le problème de ces approches est qu'elles impliquent des surcoût importants en termes de coût silicium, qui eux-mêmes induisent des niveaux de puissance statique inacceptables pour des applications de type réseaux de capteurs. Dans la section suivante, nous proposons donc une approche qui exploite la spécialisation en vue d'améliorer les niveaux de puissance dynamique dissipés, tout en contrôlant très finement le niveau de puissance statique en utilisant la technique de *power-gating*.

0.3 Une approche combinant *power gating* et spécialisation

Nous pensons qu'une approche à base de *spécialisation matérielle* offre une piste intéressante pour améliorer l'efficacité énergétique des parties calcul et contrôle embarquées dans un nœud de réseau de capteurs. Plutôt que d'exécuter l'applicatif et le système d'exploitation sur un processeur programmable, nous proposons de générer automatiquement, pour chacune des tâches du système, une micro-architecture matérielle taillée sur mesure. Une telle approche permet une réduction drastique de la puissance dynamique dissipée par chaque nœud. De plus, lorsque combinée avec des techniques de *power gating*, elle permet également de maitriser le niveau de puissance statique.

Dans notre approche, l'architecture matérielle du calculateur embarqué dans le nœud consiste en un ensemble de *micro-tâches* matérielles fonctionnant de manière concurrente, et activées en fonction de l'arrivé de tel ou tel événement.

Chacune de ces micro-tâches est chargée d'une fonctionnalité bien définie (interfaçage avec les capteurs, contrôleur MAC, routage, etc.), et est mise en œuvre sur une micro-architecture minimaliste, organisée autour d'un chemin de données dédié lui-même contrôlé par une machine à états. Cette micro-architecture est générée directement à partir d'une spécification de son comportement en C, grâce à l'adaptation d'un flot de compilation reciblable pour processeur spécialisé ASIP et d'un outil de génération de description RTL dédié à ce type d'architectures.

En combinant la spécialisation matérielle avec des techniques de réduction de puissance statique (*power gating*), nous pouvons réduire de manière très significative la puissance globale (et l'énergie) dissipée par le système [103].

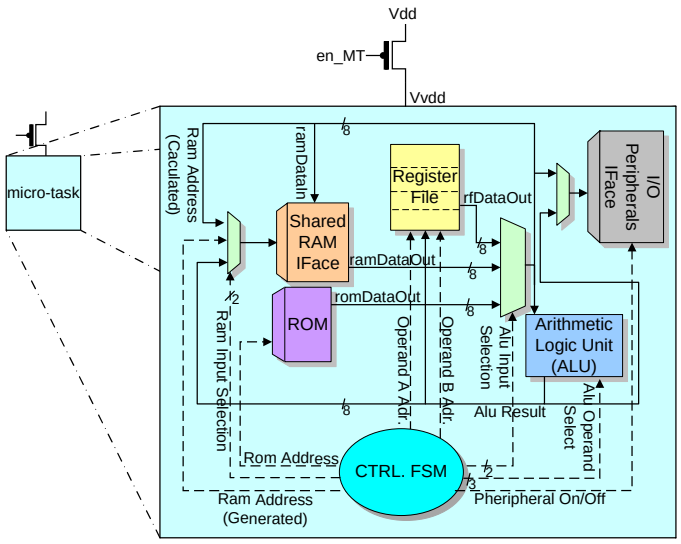


Figure 3: Architecture d'une micro-tâche matérielle generique.

0.3.1 Micro-tâches matérielles

Notre approche repose sur cette notion de micro-tâche matérielle spécialisée, qui exécute une partie des traitements du nœud. A la différence d'un processeur à jeu d'instructions, la fonctionnalité d'une micro-tâche est figée et mise en œuvre sous la forme d'une machine à états pilotant un chemin de données spécialisé. Cette mise en œuvre rend l'architecture beaucoup plus compacte (pas besoin de décodeur d'instructions, pas de mémoire de programme, etc.) et permet de dimensionner précisément tant les ressources de stockage (file de registres, ROM, RAM) que les ressources de calcul (ALU simplifiée en fonction des calculs mis en œuvre par la micro-tâche).

Chacune de ces micro-tâches peut accéder à une mémoire de données (éventuellement partagée avec d'autres tâches) ainsi qu'à des périphériques au travers d'un bus d'E/S (ex: SPI link vers un émetteur RF tel que le CC2420 [131]).

La figure 3 représente la micro-architecture d'une tâche matérielle (ici avec un chemin de données sur 8 bits). Les lignes en pointillé représentent les signaux de contrôle générés par la machine à états de contrôle, tandis que les lignes en trait continu représentent le flot de données entre les opérateurs, les ressources de stockage, etc. Une description plus détaillée de l'organisation d'une micro-tâche matérielle sera donnée en section 4.1.2.

0.3.2 Modèle au niveau système proposé

Dans cette sous-section, nous détaillons l'architecture système d'un nœud basé sur le principe de micro-tâches, dont le fonctionnement est illustré dans les paragraphes qui suivent au travers d'un exemple d'application très simple (mesure et transmission de température).

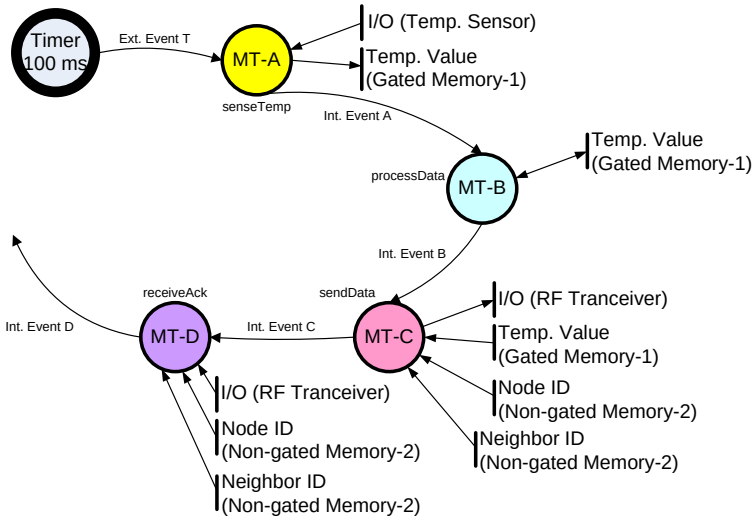


Figure 4: Graphe de tâches d'une application de relevé et envoi de température.

Graphe de tâche de l'application

Considérons une application dans laquelle nous devons : lire à intervalle régulier la mesure de température effectuée par un capteur au travers de son interface d'E/S, analyser et traiter cette valeur, l'envoyer au nœud voisin, puis enfin recevoir un acquittement de ce même voisin. Le graphe de tâche de cette application est représenté figure 4 et consiste en un ensemble de quatre micro-tâches qui echangent des données brutes et des données de contrôle.

Architecture

Le figure 5 représente une vue système d'une plateforme matérielle basée sur l'approche *micro-tâche*, et dont l'application cible (graphe de tâches) est celle proposée plus haut. Un tel système est formé:

- d'un ensemble de *micro-tâches* matérielles, contrôlées par un mécanisme de *power gating*, et qui accèdent à un ensemble de ressources partagées (RF, capteurs) et mémoires (gated/non-gated). Chacune de ces micro-tâches étant chargée d'une tâche spécifique (mesure de températuren traitement de données, etc.);

- d'un *moniteur système* (SM) qui contrôle l'activation de toutes les micro-tâches matérielles. Le moniteur système est chargé du contrôle de l'alimentation de toutes les micro-taches ainsi que des mémoires en fonction de leur utilisation;

- des périphériques capables de déclencher des événements (radio, timer, etc.) qui seront transmis au moniteur système.

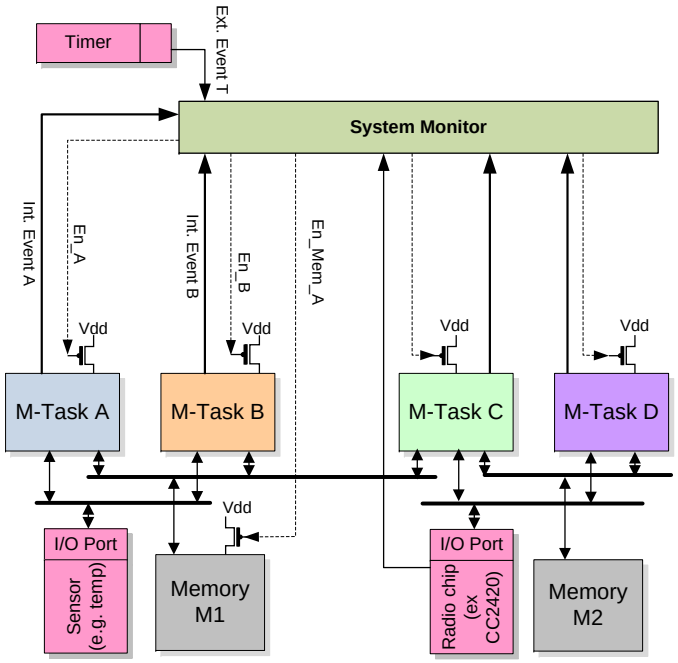


Figure 5: Vue niveau système d'un nœud de capteur basé sur l'approche à base de micro-tâches matérielles.

Fonctionnement d'un nœud de capteur basé sur l'approche micro-tâches matérielles

L'interaction entre le moniteur système (SM) et les micro-tâches matérielles reste relativement simple. Le SM échange des informations de contrôle avec chaque micro-tâche, au travers de signaux de contrôle (comme par exemple les signaux *En_A*,*En_B*, *En_Mem_1*, etc., représentés figure 1.5), de signaux d'*événements internes* (*Int.Event A*, *Int.Event B* etc., comme indiqué figure 1.5) et d'événements externes, issus des périphériques (*Ext.Event T*, etc.).

A titre d'exemple nous détaillons ci-dessous le comportement du système mettant en œuvre l'exemple du graphe de tâches de la figure 4.

- Tout d'abord le SM détecte que l'événement externe *Ext.Event T* vient de se produire, et envoie un signal de réveil à la micro-tâche *M-Task A* au travers la commande *En_A* ainsi qu'à la mémoire *Memory 1* (non alimentée à cet instant) au travers de la commande *En_Mem_1*. Une fois *M-Task A* réveillée, celle-ci va interroger le capteur de température en y accédant au travers de son bus d'E/S. Une fois lue et traité, cette valeur sera stockée dans la mémoire *Memory 1*, la micro-tâche *M-Task A* envoie alors au SM un événement interne *Int.Event A* lui indiquant qu'elle a terminé son travail.

- A la reception du signal *Int.Event A*, le SM coupe l'alimentation de *M-Task A*

en désactivant la commande *En_A*, et réveille la micro-tâche *M-Task B* qui est en charge de la seconde tâche du graphe de tâches de la figure 4, et dont le rôle est de réaliser un traitement sur la température précédemment relevée et stockée dans *Memory 1*, puis de réécrire la valeur modifiée en lieu et place de la précédente valeur en mémoire. La tâche *M-Task B* envoie alors un événement interne *Int.Event B* au SM pour lui indiquer qu'à son tour elle a terminé son travail.

- A la réception du signal *Int.Event B*, le SM coupe l'alimentation de *M-Task B* et réveille la micro-tâche *M-Task C* dont le rôle est de transmette la donnée stockée en *Memory 1* au nœud le plus proche. Pour réaliser cette tâche *M-Task C* a également besoin de la mémoire permanente *Memory 2* utilisée par le nœud pour stocker son identifiant (ID) ainsi que sa table de routage. *M-Task C* effectue alors un calcul de voisinage et envoie un paquet au plus proche voisin en accédant au composant radio par son interface SPI. *M-Task C* envoie à son tour un évenement interne *Int.Event C* au SM pour lui indiquer qu'elle a terminé son travail.

- Lorsqu'il reçoit *Int.Event C*, le MS coupe l'alimentation de *MTask C* à l'aide de la commande *En_C* et réveille la tâche *M-Task D* chargée de la réception de l'acquittement du message par le voisin. Puisque *Memory 1* n'est pas nécessaire à l'exécution de *M-Task D*, elle est également désactivée par le SM grâce à la commande *En_Mem_1* line. Finalement, une fois l'acquittement reçu par le nœud, à l'issue de l'exécution de *M-Task D*, le SM coupe l'alimentation de la quatrième et dernière tâche de l'application (sur réception de l'événement interne *Int.Event D*). De fait, l'ensemble des composants de la plateforme (à l'exception du SM et de la mémoire *Memory 2*) n'est plus alimenté. La puissance statique du système est alors réduite à son minimum.

0.3.3 Modèle d'exécution et flot de conception logiciel

Une grade partie de ce travail de thèse a porté sur le développement de l'outil *LoMiTa (ultra Low-power Micro-Tasking)*, un flot complet de conception pour plateformes matérielles dédiées [104, 105]. S'inspirant de la plupart des infrastructures pour réseaux de capteurs, ce flot se base sur l'utilisation d'un langage dédié pour la spécification système (interactions entre les tâches, gestion des événements, gestion des ressources partagées) et sur la spécification du comportement des tâche en langage C-ANSI. A partir de ces spécifications, nous sommes capables de générer du VHDL synthétisable de la plateforme dans son ensemble (micro-tâches + moniteur système), permettant une implantation directe sur ASIC ou FPGA.

Il nous semble important de préciser que notre but n'est pas de proposer un nouveau modèle de calcul pour des plate-formes de réseau de capteurs, notre approche se veut plutôt comme proposant un modèle d'exécution simple, qui soit bien adapté à ce que nous pensons être une solution architecturale innovante pour les nœuds d'un réseau de capteurs.

Nous présentons ci-dessous une vue globale de notre flot de conception à base de micro-tâches, celui-ci peut se décomposer en deux parties (c.f. figure 6):

- un outil de synthèse de matériel qui est utilisé pour générer la spécification VHDL de la micro-tâche à partir de sa spécification en ANSI-C;
- Un flot système qui se sert d'une spécification de la plate-forme et de son graphe de tâches (exprimé à l'aide d'un langage dédié) et génère la description VHDL du moniteur système.

La mise en oeuvre de notre flot de conception exploite les outils et principes du *Model Driven Engineering (MDE)*, et plus particulièrement de l'infrastructure *Eclipse Modeling Framework (EMF)* [134], ainsi que les nombreux outils et technologies qui lui sont associés.

Nous avons ainsi défini un méta-modèle pour décrire et manipuler des microarchitectures spécifiées au niveau RTL sous la forme de machine à états commandant des chemins de données (modèle *FSM+Datapath*). Ce méta-modèle est ensuite utilisé pour générer le code VHDL et SystemC des microarchitectures ainsi modélisées. En complément de ce méta-modèle, nous avons également utilisé les possibilités de l'outil MDE Xtext pour définir un langage dédié dont le but est de faciliter la spécification au niveau système de la plateforme (tâche, E/S, mémoires, etc.).

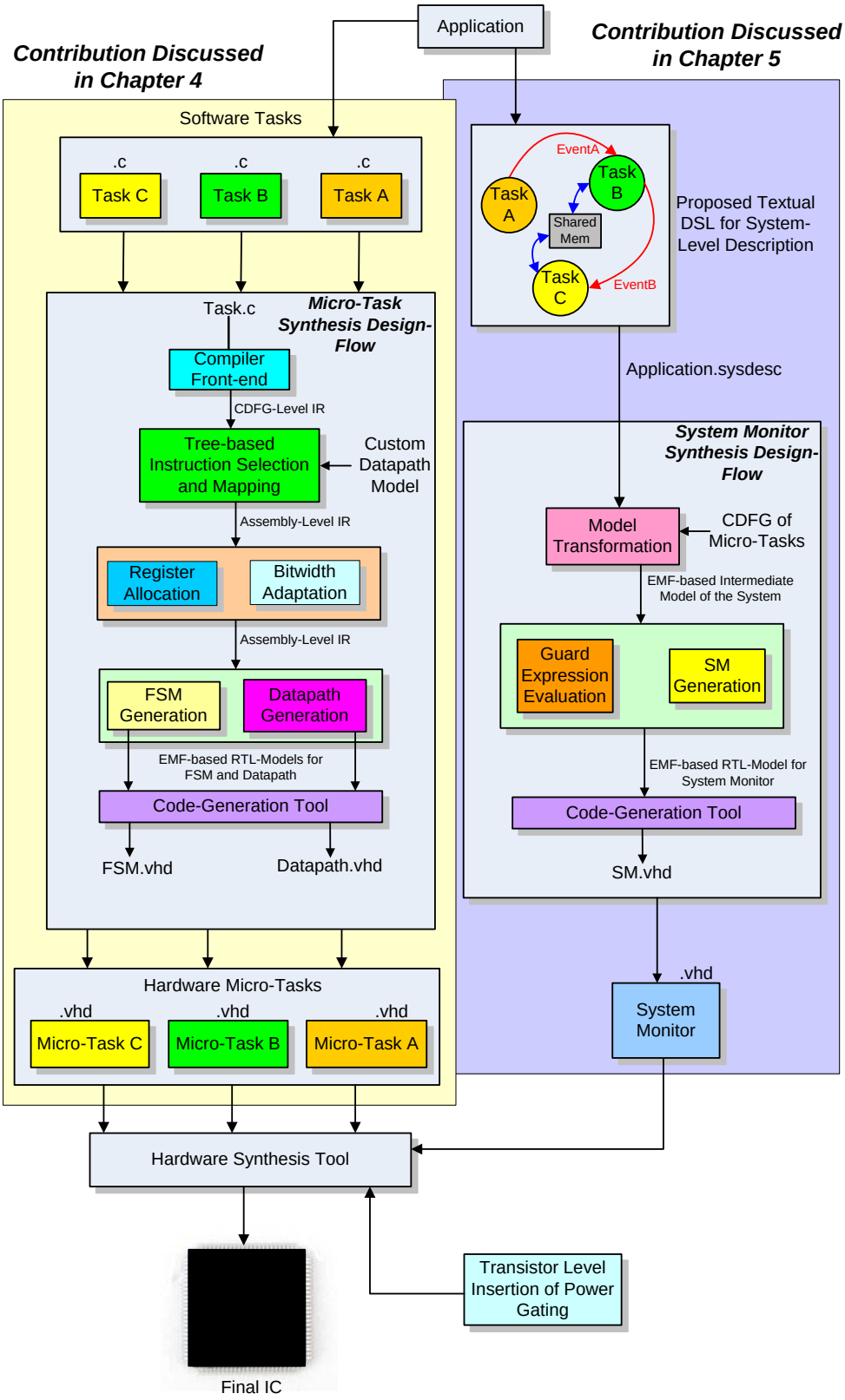


Figure 6: Flot de conception système LoMiTa

0.4 Contributions

Les contributions de cette thèse sont décrites ci dessous.

- Nous proposons un flot de conception original pour la génération de plateformes matérielles pour réseaux de capteurs très faible consommation. Ce flot se base sur la notion de micro-tâche matérielle et permet la génération d'une description d'une plate-forme complète. Dans ce flot, le comportement de chaque tâche est spécifié an C ANSI et est mappé sur une micro-architecture dédiée, grâce à une version adaptée d'un compilateur reciblable.

- Nous montrons également dans cette approche que les techniques à base de *power gating* permettent d'obtenir des temps de commutation très courts, de l'ordre de quelques dixièmes de miro-secondes, et ce même pour des micro-tâches de taille importante. Ces propriétés permettent d'améliorer le temps de réponse d'au minimum 50% par rapport à des solutions basées sur des microcontrôleurs de type MSP430.

- Nous proposons également un langage dédié (DSL) qui peut-être utilisé pour spécifier la vue système de la plateforme et qui permet de générer une description synthétisable de l'ensemble du système et en particulier le moniteur système qui est utilisée pour contrôler l'activation et la désactivation des micro-tâches matérielles.

- Notre approche permet d'obtenir des gains en puissance dynamique d'environ deux ordres de grandeur par rapport à des solution existantes à base de microcontrôleurs programmables.

- Nous avons utilisé notre flot de conception pour effectuer une étape d'exploration de l'espace de conception dans le but d'évaluer les différents compromis en surface/performance pouvant être obtenus en modifiant certains paramètres de la micro-architecture, et en particulier la largeur du chemin de données. Là encore nous avons comparé les résultats obtenus avec ceux obtenus pour une microcontrôleur comme le MSP430.

- Nous avons validé notre flot sur une application simple (mais réaliste) et montré que l'approche était tout à fait appropriée au domaine applicatif des réseaux de capteurs.

0.5 Résultats expérimentaux

Cette section présente l'ensemble des expérimentations effectuées et les résultats obtenus. Après une description des gains obtenus en termes de temps de réponse et de réveil de notre technique de *power gating* à grain fin, nous présentons les réductions de puissances dynamique et statique obtenus par notre concept de micro-tâches matérielles, en les comparant avec des implémentations à base de microcontrôleurs. Finalement, le

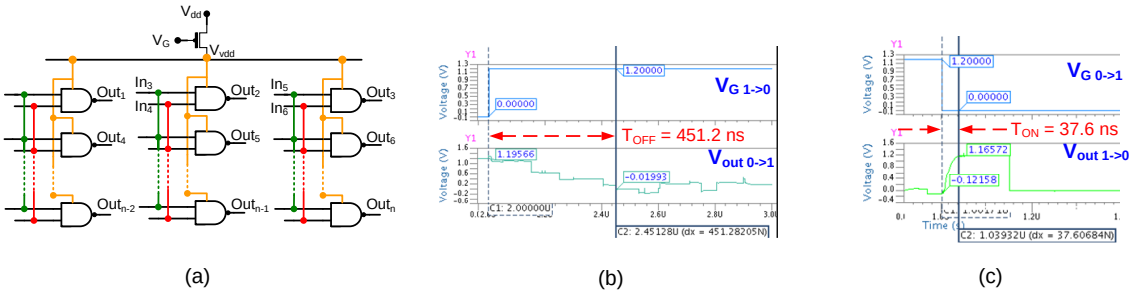


Figure 7: Modèle à base de portes NAND parallèles utilisé pour exécuter les simulations au niveau transistor à l'aide de SPICE et temps de réveil et de mise en veille mesurés pour $n = 3000$.

coût en surface et le gain en puissance statique du système complet est analysé via la synthèse du moniteur système. Tous ces résultats sont obtenus en utilisant notre flot de conception complet, tel que proposé et développé dans cette thèse.

0.5.1 Temps de commutation du *power gating*

Pour vérifier l'applicabilité du *power gating* dans l'architecture que nous proposons, nous utilisons un modèle similaire à celui utilisé par Hu et al. [59]. Cependant, comme les auteurs ne donnent pas de valeurs quantitatives des temps de commutation pour une technologie CMOS spécifique, nous avons accompli plusieurs expérimentations.

Pour cela, l'outil *Eldo* de Mentor Graphics a été utilisé pour les simulations SPICE avec une technologie CMOS 130 nm et une tension d'alimentation de 1.2 V. Nous avons utilisé un modèle à base de portes NAND parallèles (cf. figure 7 (a)). Un bloc logique de 3000 portes équivalentes, comparable en termes de surface de silicium avec la plus complexe des micro-tâches présentes dans notre système, a été simulé. Les figures 7 b et 7 c) montrent de temps d'allumage (réveil) de 37.6 ns et de coupure (mise en veille) de 451 ns entre les modes "éteint" et "actif". Ces temps doivent être comparés avec ceux du MSP430 qui offre un temps de réveil de 1 μs [129]. Ceci montre que le *power gating* est applicable dans notre système et une réduction de plus de 50% du temps de réveil peut être gagnée par rapport aux implémentations à base de microcontrôleurs.

0.5.2 Gains en puissance dynamique de l'approche à base de micro-tâches

Pour explorer les gains en consommation de notre approche, plusieurs tâches applicatives représentatives ont été extraites de benchmarks récents en réseaux de capteurs, tels que *SenseBench* [97] et *WiSeNBench* [96]. De plus, pour couvrir les applications orientées contrôle, plusieurs tâches de gestion des réseaux de capteurs dans un système d'exploitation ont été utilisées: calcul de l'adresse du prochain nœud dans un protocole de routage géographique multi-sauts (*calcNeigh*), protocole de transfert sur

Nom Tâche	MSP430						
	Nb. Instr.	Cycles Horl.	Temps (μs)	Puissance (mW)		Energie (nJ)	
				tiMSP	openMSP	tiMSP	openMSP
crc8	30	81	5.1	8.8	0.96	44.9	4.9
crc16	27	77	4.8	8.8	0.96	42.2	4.6
tea-decipher	152	441	27.5	8.8	0.96	242	26.4
tea-encipher	149	433	27.0	8.8	0.96	237.6	26
fir	58	175	10.9	8.8	0.96	96	10.4
calcNeigh	110	324	20.2	8.8	0.96	177.7	19.4
sendFrame	132	506	31.6	8.8	0.96	278	30.3
receiveFrame	66	255	15.9	8.8	0.96	139.9	15.2

Table 1: Consommation de puissance et d'énergie du MSP430 pour différentes tâches applicatives issues de benchmarks (@ 16 MHz).

bus SPI pour interfaçage avec un composant radio tel que le CC2420 (*sendFrame* et *receiveFrame*). Toutes ces tâches sont traitées via notre flot de conception qui génère les descriptions matérielles correspondantes aux micro-tâches.

Une technologie CMOS 130 nm et une tension d'alimentation de 1.2 V sont utilisées pour les résultats de synthèse. Les estimations de consommation statique et dynamique résultent d'une simulation au niveau portes à une fréquence d'horloge de 16 MHz. Les puissances estimées sont comparées avec celles dissipées par (i) tiMSP: un microcontrôleur MSP430F21x2 dont les informations sont extraites depuis la *datasheet* constructeur (8.8 mW @ 16 MHz en mode actif), ce qui inclut les mémoires et les périphériques, et (ii) openMSP, une version open-source du MSP430 (0.96 mW @ 16 MHz) synthétisée dans la même technologie 130 nm et n'incluant que le cœur et aucune mémoire ni périphérique.

Nous escomptons que la puissance dissipée réelle du cœur du MSP430 associé à sa mémoire programme se trouve entre ces deux résulats et faisons donc la comparaison avec ces deux versions.

Les résultats sont donnés dans les tableaux 1 à 3 où le tableau 1 donne le nombre de cycles et d'instructions pour les deux versions du MSP430 MCU. Les tableaux 2 et 3 montrent quant à eux les gains en puissance et en énergie obtenus par notre architecture à base de micro-tâches matérielles pour des chemins de données de 8 et 16 bits respectivement. On observe que notre approche obtient des gains en énergie entre un et deux ordres de grandeur pour les différents benchmarks.

En ce qui concerne la puissance statique, les micro-tâches consomment en moyenne 6 octets de mémoire. Quand cette mémoire est synthétisée dans une technologie 130 nm (sans optimisation spécifique), elle consomme seulement 18 nW de puissance statique. Par opposition, le MSP430 consommant approximativement 1.54 μW en statique, notre approche permet de gagner un rapport d'environ un ordre de grandeur en consommation statique par rapport aux implémentations à base de microcontrôleurs.

Nom Tâche	Micro-tâches 8-bits							
	Nb. Etats	Temps (μs)	Puissance (μW)	Energie (pJ)	Gain P. (x) P1/P2	Gain E. (x) E1/E2	Surface (μm^2)	Nb. portes Nand equiv.
crc8	71	4.4	30.09	132.4	292/32	339/37	5831.7	730
crc16	103	6.4	46.92	300.3	187/20.4	140.5/15.3	8732.5	1092
tea-decipher	586	36.6	84.5	3090	104/11.4	78/8.55	19950	2494
tea-encipher	580	36.2	87.3	3160	101/11	75/8.2	20248	2531
fir	165	10.3	75.3	775.6	116/12.8	123.8/13.4	13323.7	1666
calcNeigh	269	16.8	74.3	1248.2	118/12.9	142.4/15.5	14239.4	1780
sendFrame	672	42	33.3	1400.3	264/28.8	198.5/21.7	10578	1323
receiveFrame	332	20.7	27.3	565	322/35	247.6/26.7	5075.3	635

Table 2: Gain en puissance et en énergie pour des micro-tâches 8 bits par rapport au MSP430 (@ 16 MHz, 130 nm). P1 et E1 sont les gains en puissance et en énergie par rapport à la version tiMSP tandis que P2 et E2 sont les gains en puissance et en énergie par rapport à la version openMSP.

Nom Tâche	Micro-tâches 16 bits							
	Nb. Etats	Temps (μs)	Puissance (μW)	Energie (pJ)	Gain P. (x) P1/P2	Gain E. (x) E1/E2	Surface (μm^2)	Nb. portes Nand equiv.
crc8	71	4.4	55.3	242.6	159.6/17.4	185.1/20.2	10348	1294
crc16	73	4.56	55.0	251.0	159.8/17.4	168.1/18.3	10280	1285
tea-decipher	308	19.2	152.8	2940	57.6/6.2	82/9	27236	3405
tea-encipher	306	19.1	152.3	2910	57.8/6.3	81/8.93	27069	3384
fir	168	10.5	144.2	1514	61.02/6.7	63.4/6.9	23547	2944
calcNeigh	269	16.8	142.4	2392	61.8/6.7	74.3/8.1	24745	3094
sendFrame	672	42	58.1	2440	151.5/16.5	114/12.4	14863	1858
receiveFrame	332	20.7	50.0	1036	175.8/19.2	135/14.7	9485	1183

Table 3: Gain en puissance et en énergie pour des micro-tâches 16 bits par rapport au MSP430 (@ 16 MHz, 130 nm). P1 et E1 sont les gains en puissance et en énergie par rapport à la version tiMSP tandis que P2 et E2 sont les gains en puissance et en énergie par rapport à la version openMSP.

0.5.3 Estimation de la consommation du moniteur système

Pour comparer les consommations d'énergie et le potentiel surcoût en surface du moniteur système (SM), une description sous forme de graphe de tâches est présentée à la figure 4 et est exprimée à l'aide de notre DSL. Celui ci est ensuite traité avec notre flot de conception et une description VHDL du moniteur système qui contrôle l'activation et la désactivation des quatre micro-tâches et de la mémoire partagée est générée.

Ce code VHDL est ensuite synthétisé pour une bibliothèque de cellules CMOS standards en 130 nm afin d'obtenir les consommations statique et dynamique et le coût en surface de silicium. Les résultats montrent que le SM consomme 5.15 μW de puissance dynamique (@ 16 MHz et 1.2 V) et 296 nW de puissance statique. La partie statique peut être réduite jusqu'à 80 nW si des cellules faible consommation alimentées à 0.3 V sont utilisées pour les registres présents dans l'architecture. D'un point de vue de la surface de silicium, le SM consomme seulement 754 μm^2 (pour un graphe simple), soit environ 1% de la surface d'un cœur MSP430 synthétisé dans la même technologie.

En résumé, notre approche basée sur des micro-tâches matérielles fournit une réduction d'environ 50% dans les temps de commutation entre les modes de veille et d'activité, et des gains d'un à deux ordres de grandeur en énergie dynamique et d'un ordre de grandeur en énergie statique, par comparaison avec des implémentations logicielles sur des microcontrôleurs à très faible consommation tels que le MSP430.

Chapter 1

Introduction

1.1 Wireless Sensor Network (WSN)

Wireless Sensor Networks (WSNs) is a fast evolving technology having a number of potential applications in various domains of daily-life, such as structural-health and environmental monitoring, medicine, military surveillance, robotic explorations etc. Advancements in Micro-Electro-Mechanical-Systems (MEMS) technology, wireless communications, and digital electronics have facilitated the development of low-cost, low-power, multi-functional sensor nodes that are small in size and communicate efficiently over short distances. Thus the emerging field of WSN combines sensing, computation, and communication into a single tiny device (WSN node). WSN Systems of 1000s or even 10,000s of such nodes are anticipated that can revolutionize the way we live and work.

The core design challenge in WSN is coping with the harsh resource constraints placed on the individual node devices. Embedded processors controlling the WSN nodes have only kilo-Bytes of memory and they must implement complex networking protocols. Many constraints evolve from the fact that these devices will be produced in a large number and must be small and inexpensive. As Moore's law still remains applicable, we get nearly double the number of transistors in same surface area with newer process technology. This results in two scenarios (i) more functionalities can be added to a device for the same given area or (ii) size of the device gets smaller for the same given functionalities. This size reduction is also helpful for the devices to be produced as inexpensively as possible.

The most difficult resource constraint to meet is power consumption. As physical size decreases, so does energy capacity of a WSN node. Underlying energy constraints end up creating computational and storage limitations that lead to a new set of architectural issues. Many devices, such as cell phones and pagers, reduce their power consumption through the use of specialized communication hardware in Application Specific Integrated Circuits (ASICs) that provide low-power implementations of the necessary communication protocols.

To date, the strength of WSN systems is supposed to be their flexibility and uni-

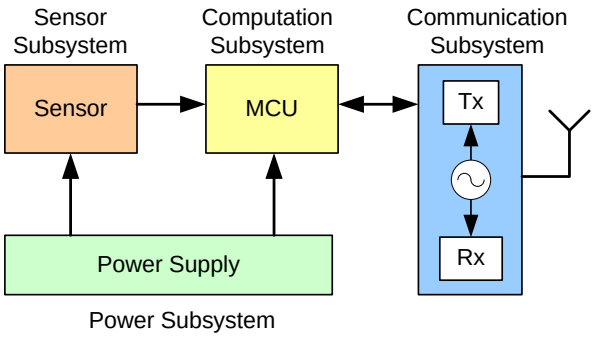


Figure 1.1: General architecture of a WSN node.

versality. However, when looking more carefully to actual design practices, we observe that the need for flexibility/programmability is essentially geared toward the user application layer, which happens to represent only a small fraction of a WSN node's processing workload. Whereas most of the processing workload is almost dedicated to the communication protocol stack. Hence, in our opinion, it is worth-studying to explore the hardware specialization approach in WSN node design as well to meet the ultra low-power requirement. In order to reduce the power consumption in a WSN node, we first need to look at the generic node architecture to find out the hotspots for power consumption. The generic architecture of a WSN node is discussed in the next section.

1.1.1 WSN node architecture

WSN nodes are low-power embedded devices consisting of processing and storage components (a Microcontroller Unit (MCU) connected to a RAM and/or flash memory) combined with wireless communication capabilities (RF transceiver) and some sensors/actuators. Since these nodes must have small form-factors and limited production cost, it is not possible to provide them with large energy sources [138]. In most cases they must rely on non-replenishing (e.g. battery) or self-sufficient (e.g. solar cells) sources of energy.

Figure 1.1 presents the system architecture of a generic sensor node. It is composed of four major subsystems: power supply, communication, control and computation, and sensing. The power supply subsystem consists of a battery and a DC-DC converter and has the purpose to power-up the node. The communication subsystem consists of a radio transceiver for wireless communication. Most of the platforms use a single omni-directional antenna however, cooperative "Multiple-Input and Multiple-Output (MIMO)" technology has also been deployed [99]. The processing subsystem is typically composed of memory to store application program codes and data, and of a microcontroller to control the system and process the data. The last subsystem links the sensor node to the region of interest and has a group of sensors and actuators that depend on the WSN application. It also has an Analog-to-Digital Converter (ADC) to

convert the analog data sensed by the sensors to digital data that can be used by the processing subsystem. To design such architecture with limited resources, the designers are faced with some tough constraints that are discussed in the following section.

1.1.2 WSN node design constraints

Designing a WSN node is a challenging task, since the designers must deal with many stringent design constraints and metrics that are often interrelated. Here we will briefly discuss some of the metrics that are considered while designing a WSN node.

- ***Power*** is the biggest design challenge to meet while designing individual sensor nodes to implement the applications that require multi-year life-time.
- ***Robustness*** also becomes an important parameter in WSN node design to support correct functioning of the network. Each node must be designed to be as robust as possible to tolerate and adapt to neighborhood node failures.
- ***Security*** at application-level is another metric to be considered while designing a node. The individual nodes must be capable of performing relatively complex encryption and authentication algorithms.
- ***Communication bit-rate and range*** are key design metrics for a WSN node as well. An increase in the communication range (and bit-rate) has a significant impact on the power consumption (and computational requirement) of the node.
- ***Computation workload*** is another key design metric and it directly influences a node's power consumption. The more a node would be computationally-intensive, the more would be its overall power/energy budget.
- ***Cost and size*** The physical size and cost of each individual sensor node has a significant and direct impact on the ease and cost of deployment as well as the size of the energy source available to it.

As discussed earlier, since WSN nodes are deployed in huge numbers, they must be of small form-factor and inexpensive. Besides, it is not possible to equip them with large power sources. Hence, ultra low-power becomes the most critical design metric for a WSN node. It is also supported by the fact that WSN nodes may have to work unattended for long durations due to a large number of deployed nodes or a difficult access to them after deployment.

If we analyze the power profile of a WSN node, among all the subsystems (Section 1.1.1), communication and computation subsystems consume bulk of a node's available power-budget [116, 30]. In this work, we are targeting the power optimization of the computation and control subsystem of a WSN node. Indeed, we believe that power and energy savings obtained through our approach could open possibilities for more computationally demanding protocols or modulation schemes which, as a result, would provide better Quality-of-Service (QoS), lower transmission energy and higher

network efficiency. Power reduction for computation and control subsystem can benefit from optimizations at several levels of a WSN node design (such as application design, micro-architecture design, logic synthesis and Very Large Scale Integrated (VLSI) circuit design).

We will discuss the power reduction techniques adapted at two different levels of a WSN node design, i.e. micro-architectural level and VLSI circuit level as they are the two levels targeted by our approach.

1.2 Power optimization of a WSN node

In the last decade, there have been a large number of research results dealing with power optimization in VLSI circuits. A lot of research has also been done to optimize power at micro-architectural level such as evolution of low-power MCUs. This section briefly covers some of these power optimization techniques.

1.2.1 Low-power VLSI design

Power dissipation in VLSI circuits can be divided into two categories: *dynamic power* caused by capacitance switching (i.e. stage changes) that occurs while a circuit is operating and *static power* caused by leakage current between power supply and ground. When a device is active, its power is usually largely dominated by dynamic power and becomes roughly proportional to clock frequency. However, in the context of WSN, things are slightly different as a WSN node remains inactive for long periods (MCU duty-cycle lower than 1%), and the contribution of static power also becomes significant and can not be ignored.

There are several approaches to reduce dynamic power in a circuit (e.g. clock gating, voltage scaling etc.) that can be applied at various levels of the design-flow. However, most of them are poorly suited to WSN nodes as they often significantly increase the total silicon area, and therefore have a negative impact on static power dissipation.

One exception is *power gating*, which consists in turning-off the power supply of inactive circuit components [12, 87]. *Power gating* helps in reducing both dynamic and static power, and is thus very efficient for devices in which components remain idle for long time periods.

The technique consists in adding a *sleep transistor* between the actual V_{DD} (power supply) rail and the component V_{DD}, thus creating a *virtual supply voltage* called V_{VDD} as illustrated in Figure 1.2. This sleep transistor allows the supply voltage of the block to be cut off to dramatically reduce leakage currents.

1.2.2 Low-power MCUs

As far as the power optimization at micro-architectural level is concerned, a number of low-power microcontrollers (e.g. MSP430, CoolRISC and ATmega128L) have been designed that share several characteristics: a simple datapath (8/16-bit wide), a reduced number of instructions (only 27 instructions for the MSP430), and several power saving

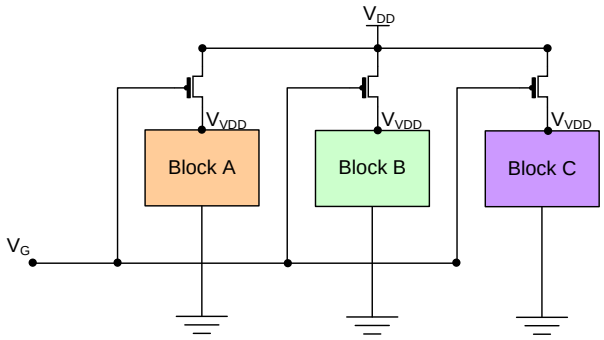


Figure 1.2: An example of power gating.

modes which allow the system to select at run-time the best compromise between power saving and reactivity (i.e. wake-up time). These processors, are designed for low-power operation across a range of embedded system application settings but, are not necessarily well-suited to the event-driven behavior of WSN nodes as they are based on a general purpose, monolithic compute engine.

Most of the current WSN nodes are built on these commercial MCUs. For example, Mica2 mote [25] has been widely used by the research community and is based on ATmega128L from Atmel. The same MCU has also been used by the designers of the eXtreme Scale Mote (XSM) [30]. The Hydrowatch [39] and PowWow [64] platforms are built on the MSP430 [129] from Texas Instruments whereas the WiseNet platform from CSEM uses a CoolRISC-core [33] from EM Microelectronic.

Although the power consumptions of these MCUs may seem extremely small w.r.t. the power budgets of typical embedded devices, looking at energy efficiency metrics such as *Joules per Instruction*, it appears that the proposed architectures (such as the MSP430) still offer room for improvement. In particular, it is clear that a combination of specialization and parallelism would significantly help improving energy efficiency. However, such architectural improvements usually come at a price of significant increase in silicon area, which leads to unacceptable levels of static power dissipation for WSN. In the next section, we will discuss our proposed approach that exploits the hardware specialization technique to improve the dynamic power consumption and also tackles the issue of increased static power dissipation using power gating.

1.3 Proposed approach: combination of power-gating and hardware specialization

We believe that the *hardware specialization* is an interesting way to further improve energy efficiency in WSN computation and control subsystem i.e. instead of running the application and control tasks on a programmable processor, we propose to generate an application specific micro-architecture, tailored to each task of the application at hand. This approach results in a drastic reduction of dynamic power dissipation of

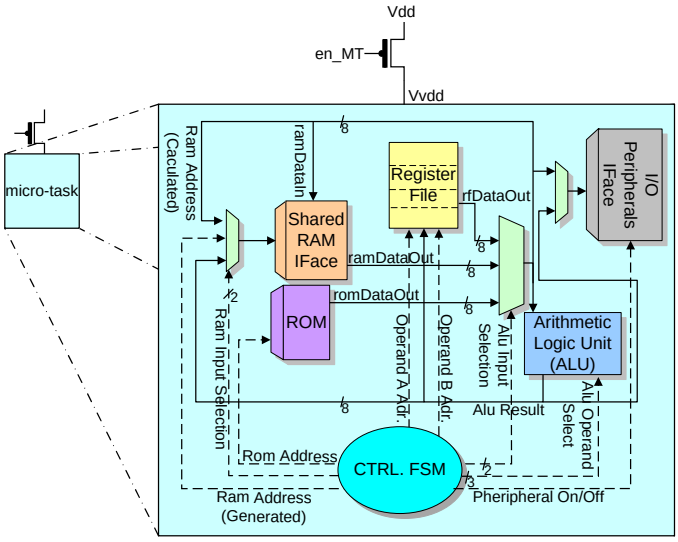


Figure 1.3: Architecture of a generic hardware micro-task.

a WSN node. On the other hand, to tackle the (potential) increase in static power consumption, we propose to use *power gating* technique.

We propose such an approach where a WSN node architecture is made of several hardware *micro-tasks* that are activated on an event-driven basis, each of them being dedicated to a specific task of the system (such as event-sensing, low-power MAC, routing, and data processing etc.). The architecture of a hardware micro-task is in the form of a minimalistic datapath controlled by a custom Finite State Machine (FSM) and is being automatically generated from a task specification in C, by using an Application Specific Instruction-set Processor (ASIP)-like design environment retargeted to our purpose.

By combining hardware specialization with static power reduction techniques such as *power gating*, we can drastically reduce both dynamic (thanks to specialization) and static (thanks to power gating) power [103].

1.3.1 Power-gated micro-task

Our approach relies on the notion of a specialized hardware structure called a hardware *micro-task*, which executes parts of the WSN node code. In contrast to an instruction-set processor, the program of a micro-task is hardwired into an FSM that directly controls a semi-custom datapath. This makes the architecture much more compact (neither an instruction decoder is needed, nor an instruction memory) and allows the size of storage devices (register file and RAM/ROM) as well as the Arithmetic Logic Unit (ALU) functions to be customized to the target application. Each of these micro-tasks can access a shared data memory and peripheral I/O ports (e.g. SPI link to an RF transceiver such as the CC2420 [131]).

Figure 1.3 shows the micro-architecture for such a hardware micro-task (here with

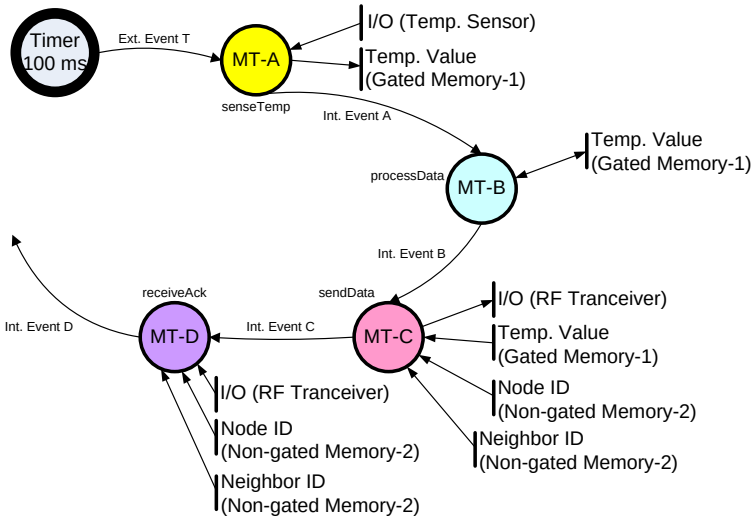


Figure 1.4: TFG of a temperature sensing and forwarding application.

an 8-bit data-path), dotted lines represent control signals generated by the control FSM, whereas solid lines represent data-flow connections between datapath components. The details of the micro-task architecture are given in Section 4.1.2.

1.3.2 Proposed system model

The basic system architecture of a WSN-node based on micro-task-oriented approach and its behavior is explained ***with the help of*** a simple temperature sensing and forwarding application in the following paragraphs.

Application task graph

Consider, for example, an application in which we periodically read the temperature value provided by a temperature sensor through I/O interface, process this value, send it to a neighbor node and finally, receive an acknowledgment from that neighbor node. The task flow graph (TFG) of this application is shown in Figure 1.4 and consists of a set of 4 micro-tasks and some data and control communication.

Architecture

Figure 1.5 represents the system level view of a WSN node platform designed according to our proposed approach to implement the above-mentioned TFG. Such a system consists of:

- A set of power-gated hardware micro-tasks accessing shared resources (e.g. peripherals (RF, sensor) and memories (gated/non-gated)). Each of these hardware micro-tasks is able to perform a specific task such as temperature sensing, data processing etc.

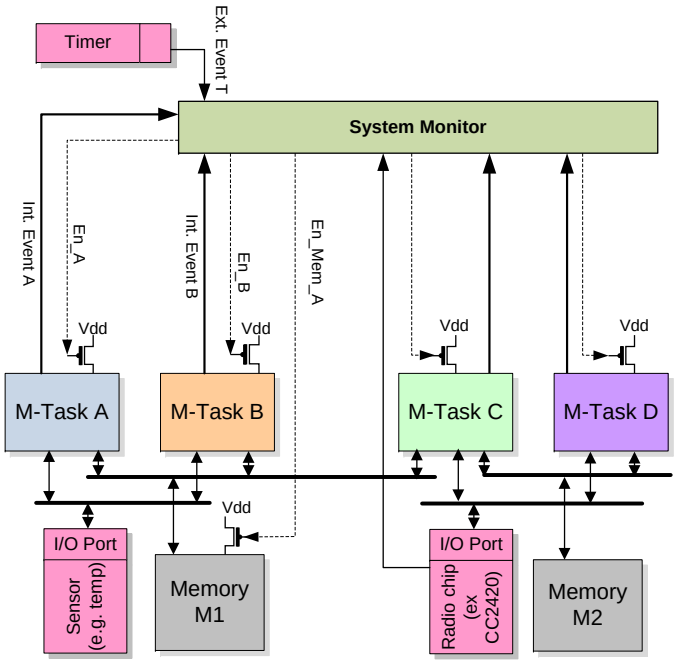


Figure 1.5: System-level view of a micro-task based WSN node architecture.

- A hardware *System Monitor* (SM) that controls the execution of all the hardware micro-tasks. The SM is responsible for the turning-ON/OFF of the hardware micro-tasks as well as the power-gated memories depending upon their usage.
- Event triggering peripherals (such as radio, timer etc.) that can send events to the SM.

Behavior of a micro-task-based WSN node

The behavior of a WSN node based on our approach is simple. There are control signals exchanged between the hardware micro-tasks and the SM. The control signals generated by the SM are called "commands" (e.g. *En_A*, *En_B*, *En_Mem_1* etc.) as shown in Figure 1.5, whereas signals emitted by the hardware micro-tasks after their job-completion are called "internal events" (e.g. *Int.Event A*, *Int.Event B* etc.) shown in Figure 1.5. There are also some events generated by the I/O peripherals that are called "external events" (e.g. *Ext.Event T* etc.).

The behavior of the WSN node, implementing the TFG example shown in Figure 1.4 is explained below:

- The SM receives an external event *Ext.Event T* and through combinational logic implemented in it, it sends the wake-up signal to the power-gated micro-task *M-Task A* through *En_A* and the power-gated memory *Memory 1* through *En_Mem_1*.

M-Task A wakes up and execute the temperature sensing task by reading the temperature value from temperature sensor through I/O peripheral. This data is then stored in *Memory 1*. Then *M-Task A* sends an internal event *Int.Event A* announcing the SM that it has finished its job.

- Upon receiving *Int.Event A*, the SM shuts down the *M-Task A* by power-gating its supply-voltage through *En_A* and wakes up the hardware micro-task *M-Task B* implementing the second task present in the TFG of Figure 1.4. This hardware micro-task reads the temperature value stored in *Memory 1*, processes it accordingly and stores the new value back in the memory. *M-Task B* then sends an internal event *Int.Event B* to the SM announcing its job-termination.

- When *Int.Event B* is read by the SM, it shuts down the *M-Task B* by power-gating *En_B* and wakes up *M-Task C* that sends the post-processed data stored in *Memory 1* to the nearest neighbor. To perform this task, *M-Task C* also needs the non-power-gated memory *Memory 2* that is being used by the node to store its node ID and the neighborhood table. *M-Task C* performs a simple neighborhood calculation and sends the packet to the nearest neighbor by writing it to the SPI interface of the RF transceiver. *M-Task C* then sends an internal event *Int.Event C* to the SM announcing the termination of its job.

- Upon receiving *Int.Event C*, the SM powers down the *MTask C* by power-gating its supply-voltage through *En_C* and wakes up the hardware micro-task *M-Task D* implementing the reception of acknowledgment from the neighbor. Since *Memory 1* is not needed by *M-Task D*, it is also power-gated by SM through *En_Mem_1* line. Finally, once the acknowledgment is received by the node, through successful finish of *M-Task D*, the SM shuts down the fourth and the last hardware micro-task present on the WSN node platform at the reception of an internal event *Int.Event D*. Resultantly, the whole computation and control part of the WSN node, except the SM and *Memory 2*, goes to power-off mode to conserve power.

1.3.3 Customized execution model and software design-flow

Major portion of this work is devoted to the development of *LoMiTa (ultra Low-power Micro-Tasking)*, a complete system-level design-flow for designing application specific hardware platforms [104, 105]. Following the philosophy of many WSN software frameworks, this flow uses a combination of a textual Domain Specific Language (DSL) for system-level specifications (interactions between tasks, event management, shared resources management etc.) and ANSI-C for specifying the behavior of each micro-task. From such a specification, we are able to generate a synthesizable VHDL (VHSIC Hardware Description Language; VHSIC: Very-High-Speed Integrated Circuit) description of the whole architecture (micro-tasks and SM), which provides a direct path to ASIC or FPGA (Field Programmable Gate Array) implementation.

We want to clarify that our goal is not to propose a new model of computation for WSN computation subsystem. We rather see our approach as a simple execution

model chosen so as to be a good match for what we think is a promising architectural solution for WSN nodes.

Here we provide a brief overview of our software framework for designing micro-task based WSN platforms. It consists of two parts:

- A customized ANSI-C to hardware compiler which is used to generate the VHDL specification of a micro-task, given its behavior in ANSI-C.
- A design-flow that uses a system specification in DSL and generates a VHDL description for its hardware SM.

Our complete design-flow is summarized in Figure 1.6: we start from the application task descriptions written in ANSI-C and a system-level description of task interactions described in a textual DSL to derive the behavior of the SM and close the path to hardware generation.

The design-flow is based on different Model Driven Engineering (MDE) techniques. To briefly elaborate the concept: the micro-task generation flow is based on Eclipse Modeling Framework (EMF) [134] that is used to define the Register Transfer Level (RTL) EMF-models for FSM and datapath components whereas the SM generation flow is based on Xtext [136] that is used to develop a textual DSL to describe different components of the system-model like micro-tasks, their corresponding events, shared memories and I/O resources. Then the flow uses this description to generate the RTL EMF-models for the SM components. Both of the design-flows, then use the code-generation techniques to generate the VHDL descriptions for the hardware micro-tasks and the SM architectures.

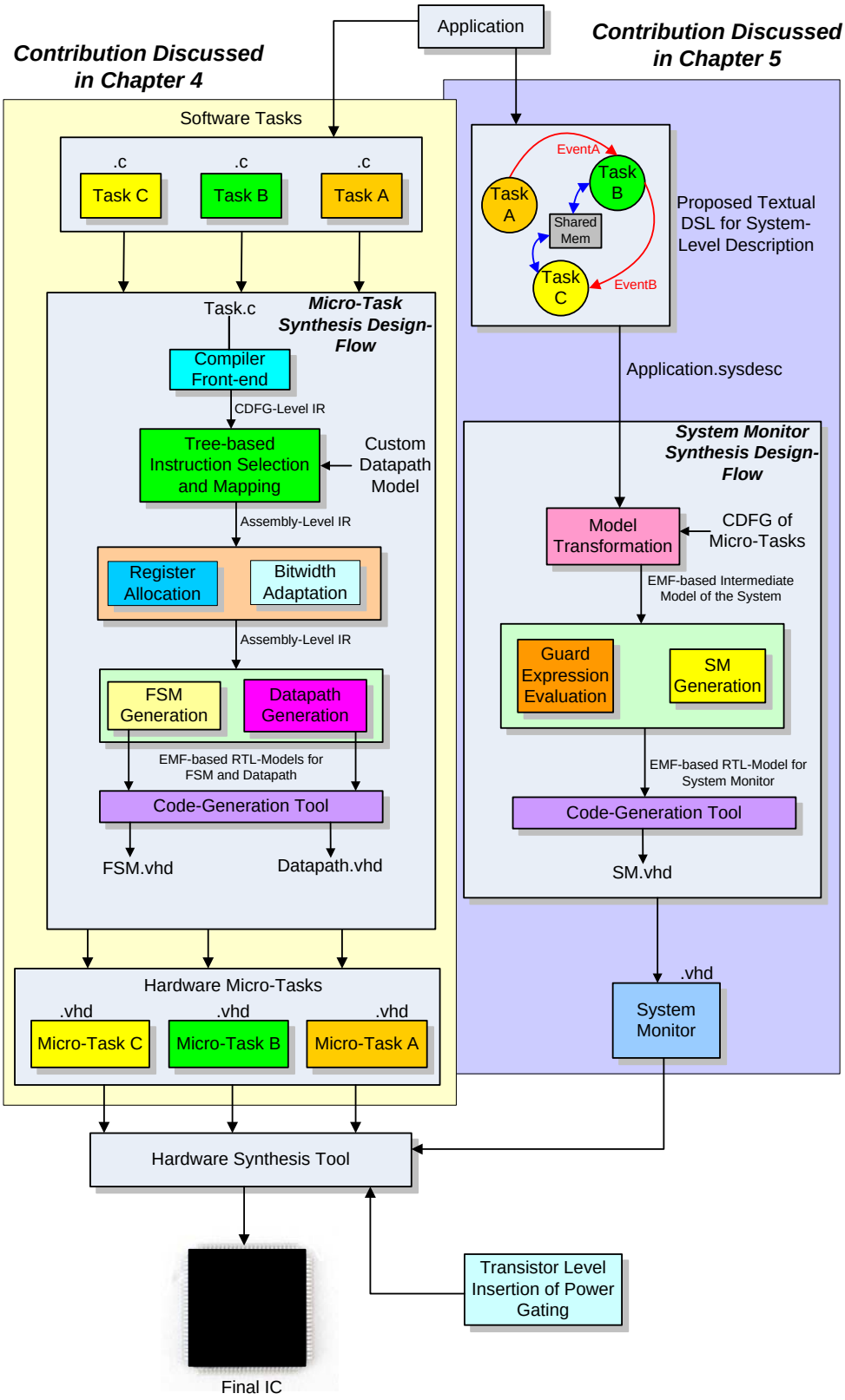


Figure 1.6: Complete system-level design-flow

1.4 Contributions

The main contributions of this work can be described as under:

- We provide an integrated design-flow for micro-task-based WSN node controller synthesis. In this flow, the behavior of each micro-task is specified in C and is mapped to an application specific micro-architecture using a modified version of a retargetable compiler infrastructure.
- We also show that the power-gating scheme happens to have very short switching-time delays, in the orders of a few hundred of nano seconds for average-size hardware micro-task designs. This improves the wake-up response time by at least 50 % when compared to low-power MCUs such as the MSP430.
- We also provide a DSL that can be used to specify the system-level execution model of a WSN node that in turn generates the hardware description for the system monitor (SM) that is used to control the activation and deactivation of the power-gated hardware micro-tasks.
- Our approach provides power savings of one to two orders of magnitude in dynamic power when compared to the power dissipation of currently available low-power MCU-based solutions.
- We also use this flow to perform design space exploration by exploring the trade-offs in power/area that can be obtained by modifying the bitwidth of the generated hardware micro-tasks. We compare the obtained results to those achieved by using an off-the-shelf low-power MCU such as the MSP430.
- We performed realistic case-study of a WSN application that serves as an experimental validation that the approach is conceivable for real-life WSN systems.

1.5 Thesis organization

The thesis is organized in 7 chapters. *Chapter 2* presents the related work about power optimization of WSN node architecture. It starts by briefly covering the basics of WSN systems. It then describes the generic components of a WSN node and their contribution to the node's power budget. *Chapter 2* covers the commercially available WSN node architectures and provides a short survey of their features and design parameters. It provides a summary of power reduction techniques developed at different design levels of a WSN node such as micro-architectural level and VLSI circuit level.

Since our design-flow for micro-task generation is based on a hybrid of High-Level Synthesis (HLS) and ASIP design methodologies, *Chapter 3* provides a survey of the existing work done in both of these domains. It starts with generic HLS design methodology and covers the existing tools for HLS. Similarly, it also discusses the ASIP design-flows and existing tools.

Chapter 4 thoroughly describes our proposed design-flow for hardware micro-task synthesis. It starts by explaining the notion of a hardware micro-task, the basic building block of our proposed approach. It discusses its potential power benefits and generic architecture. It then concludes with a comprehensive description of our proposed micro-task synthesis design-flow and a small experimental demonstration.

Chapter 5 covers the second half of our design tool, *LoMiTa*, that is the development of a system-level execution model and design-flow for the hardware SM synthesis. It starts by a brief introduction to existing execution paradigms in embedded systems and why event-driven approach is more suitable for WSN systems. It then covers the system-level view of the computation and control part of a WSN node based on our approach. It then adds a comprehensive survey of existing WSN-specific OS that are used for task- and power-management in conventional WSN nodes. *Chapter 5* summarizes the features of our proposed system-level execution model afterward and finally concludes with the details of the System Monitor (SM) synthesis design-flow and a simple example demonstrating the power benefit and area overhead of a hardware SM.

Chapter 6 consists of the experimental setup and the results that we have achieved. It starts by describing the effects of using power-gating technique in our system and achieved improvement in wake-up response time. It then covers the dynamic and static power reductions achieved by our approach as compared to the currently available low-power MCUs in the light of a case study WSN application. Additionally, it provides the findings based on the design space exploration that we performed by varying the sizes and bitwidths of the micro-task components and summarizes an optimal option. *Chapter 6* also provides the result for the power consumption of the SM controlling the micro-tasks of our case-study example and compares it with an MCU-based software solution. It concludes with the expression of overall energy gain for a complete time-period of a micro-task activation.

Chapter 7 concludes the work done in the thesis along with the international publications extracted from this work and draws some future research directions.

Chapter 2

WSN node architectures and low-power microcontrollers

2.1 WSN basics

WSN systems are a merger of wireless networks and sensors but have different features and design challenges from both of them and provide possible significant improvements over both of them. For instance, traditional sensors are generally deployed in two ways [65]:

- Sensors are placed far from the region of interest. In this approach, large sensors with complex measuring techniques and algorithms are required that must be capable of distinguishing the data readings from the environmental noise.
- Several sensors are deployed within the area of interest but they only perform the sensing. Hence, the position and communication topologies for such sensors are carefully designed. They transmit periodically sensed readings that are combined and processed at central nodes.

In contrast, a WSN is composed of a large number of sensor nodes that are deployed either inside the region of interest or very close to it. The positioning topology of the nodes does not need to be pre-designed. This makes the deployment simpler and helps in certain applications such as random deployment during disaster relief operations. On the other hand, it means that sensor nodes must be equipped with self-organization and localization functions. Moreover, sensor nodes contain on-board processing units and they do not just perform simple sense and forward operation but use their processing capabilities to perform *in-network* data processing and fusion.

As far as the comparison between traditional wireless networks and WSN is concerned, we highlight the distinguishing features and design challenges for WSN below (more details can be found in the article by Holger et al. [70]):

- ***Multi-hop communication***: While in traditional wireless networks both single-hop and multi-hop communications are feasible depending upon the application

requirements, in WSN mostly multi-hop communication is preferred. In particular, communication over long distances requires a high transmission power that is not possible for the allowed energy-budget of a WSN node. Hence, the use of intermediate nodes as relays can reduce the overall required energy of the system.

- ***Energy-efficient operation***: To support long life-times, energy-efficient operation is a key technique. Since the WSN nodes must be small-size and cost-effective, they can not be equipped with huge energy sources [138]. Hence, the design of a WSN node must ensure the energy-efficient computation (measured in *Joules per instruction*) and energy-efficient communication (measured in *Joules per transmitted bit*) and find, wherever possible, the best compromise between the two operations.

- ***Unattended operation***: Since WSN nodes can be deployed in huge numbers (1000s to 10,000s) and moreover in inaccessible terrains, it is not possible to perform their maintenance after deployment. Hence, they must be robust and autonomous in their configuration and energy needs.

- ***Collaboration to in-network processing***: In some applications, a single WSN node is not able decide whether an event has happened but several sensors have to collaborate to detect an event. This collaboration results in data-aggregation of the readings as they propagate forward through the network, reducing the amount of data to be transmitted and hence an improvement in overall energy consumption of a WSN.

- ***Fault tolerance***: Sensor nodes may fail or be blocked due to lack of power, physical damage or interference. The failure of some sensor nodes should not affect the overall system operation and a WSN must be designed with reliability or fault tolerance capabilities.

Moreover, as mentioned earlier that a WSN consists of a relatively large number of sensor nodes, hence the physical size and cost of each individual sensor node has a significant and direct impact on the ease and cost of deployment as well as the size of the energy sources available to them. Hence, we can clearly see that having a low-power design is the basic and most important driving force behind the engineering of a WSN node. In the next section, we will have a closer look at the generic architecture of an individual WSN node architecture as discussed in the literature.

2.2 WSN node architectures

When designing a WSN node, evidently the application requirements play a decisive factor with regard mostly to size, costs, and energy consumption of the nodes, but the trade-offs between features and cost is crucial. In some extreme cases, an entire sensor node should be smaller than $1\,\mathrm{cm}^3$, weigh (considerably) less than $100\,\mathrm{g}$, be substantially cheaper than US$ 1, and dissipate less than $100\,\mu\mathrm{W}$ [115]. In even more

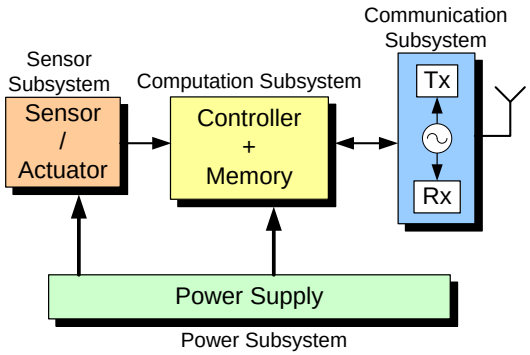


Figure 2.1: A generic WSN node architecture.

extreme visions, the nodes are sometimes claimed to have to be reduced to the size of grains of dust [138]. In more realistic applications, the mere size of a node is not so important; rather, convenience, simple power supply, and cost are more important [17].

A basic sensor node is comprised of four main components (as shown in Figure 2.1) that are discussed in the following sections.

2.2.1 Computation subsystem

The computation subsystem is the core of a wireless sensor node. It gathers data from the sensors, processes this data, decides when and where to send it, receives data from other sensor nodes, and activates the actuator accordingly. It has to execute various programs, ranging from time-critical signal processing and communication protocol stack to application programs. It can very-well be called the Central Processing Unit (CPU) of the node.

In most of the currently developed WSN node architectures, computation subsystem includes a general purpose low-power embedded MCU and in certain cases it can also contain coprocessing elements (such as hardware accelerators). As far as the storage part is concerned, a variety of storage devices are deployed such as Random Access Memory (RAM) to store the temporary data being received or processed by the node, Read Only Memory (ROM) or Flash memory to store the permanently needed data like node ID, neighborhood node table, etc. In Section 2.5, we will discuss about some of the low-power MCUs extensively used in commercial and academic WSN node architectures.

2.2.2 Communication subsystem

For wireless communication subsystem of a node the usual choices include Radio Frequency (RF), optical communication, and ultrasound. Of these choices, RF-based communication is by far the most extensively used as it best fits the requirements of most WSN applications. For instance, it provides relatively long range and high data rates, acceptable error rates at reasonable energy expenditure, and does not require

Measurement for Wireless Sensor Networks		
	Measured	**Transduction principle**
Physical properties	Pressure	Piezoresistive, capacitive
	Temperature	Thermistor
	Humidity	Resistive, capacitive
	Flow	Pressure change
Motion properties	Position	GPS, contact sensor
	Velocity	Doppler, Hall effect
	Acceleration	Piezoresistive, piezoelectric
Presence	Tactile	Contact switch, capacitive
	Proximity	Hall effect, magnetic, seismic
	Distance	Sonar, radar, magnetic
	Motion	Sonar, radar, acoustic, seismic (vibration)

Table 2.1: Some measured quantities and corresponding physical principles used to measure them.

Line of Sight (LoS) between sender and receiver. Some examples of RF-based radio transceivers are CC 1000 [132] and CC 2420 [131] from Texas Instruments, and TR 1000 from RFM [120].

2.2.3 Sensing subsystem

WSN nodes may consist of many different types of sensors such as seismic, low sampling rate magnetic, thermal, visual, infrared, acoustic and radar which are able to sense a wide variety of environmental conditions. Table 2.1 summarizes some of the physical principles that may be used to measure various quantities (as indicated by Lewis [38]).

Actuators are just about as diverse as sensors, yet for the purposes of designing a WSN, they are a bit simpler to take account of: in principle, all that a WSN node can do is to open or close a switch or a relay or to set a value in some way. Whether this controls a motor, a light bulb, or some other physical object, is not really of concern to the way communication protocols are designed.

2.2.4 Power supply subsystem

The power supply subsystem consists of a power supply and (possible) DC-DC converter. Power supplies can be of different types but they mostly lie in two major categories: (i) Non-replenishing: mostly consisting of simple batteries and (ii) Self-sufficient: consisting of energy-scavenging mechanism (e.g. solar (photo-voltaic) cells, Piezoelectric and thermal-gradient materials etc.) to harvest the energy and batteries to store it.

Moreover, batteries (or other forms of energy storage) alone are not sufficient as a direct power source for a sensor node. One typical problem is the reduction of battery voltage as its capacity drops. This can result in less power delivered to the sensor node circuits, with immediate effects on oscillator frequencies and transmission power. Hence, a DC-DC converter is used to overcome this problem by regulating the voltage delivered to the node circuitry. However, to ensure a constant voltage even

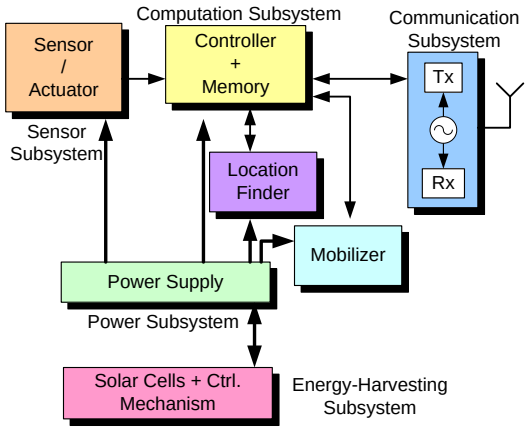


Figure 2.2: Block diagram of a mobile sensor WSN node.

though the battery output voltage drops, the DC-DC converter has to draw increasingly higher current from the battery when the battery is already becoming weak, speeding up battery death [116]. Also, the DC-DC converter does consume energy for its own operation, reducing overall efficiency. However, the advantages of predictable operation during the entire life-cycle of the node can outweigh these disadvantages.

In addition to the four basic subsystems discussed-above, a WSN node can contain additional application-specific subsystems. These additional subsystems can be location-finding and mobilization subsystems in a mobile sensor node or an energy-harvesting subsystem that works in cooperation with the power-supply subsystem to create an autonomous sensor node w.r.t. the energy needs [5]. Figure shows a scenario where these additional subsystems are included to create a energy-harvesting mobile WSN node.

In the next section, we will discuss the power profile and major sources of power dissipation in a WSN node architecture.

2.3 Power dissipation analysis of a WSN node

As mentioned earlier, in some applications, replenishment of power sources might be impossible. Sensor node life-time, therefore, is strongly dependent on battery life-time. In a multi-hop sensor network, each node plays a dual role, both of data originator and data router. As a consequence, few node failures can cause significant topological changes and might result in re-routing of packets and re-organization of the network. Hence, power consumption and power management take on additional importance. For these reasons, both power-aware software (such as protocols and algorithms) and hardware design are the current focus of WSN research community.

The main task of a sensor node in a sensor field is to detect events, perform quick local data processing, and then transmit the data. Power consumption can hence be divided into three domains: sensing, communication, and data processing.

Sensing power varies with the nature of applications. Sporadic sensing might consume lesser power than constant event monitoring. However, sensing subsystem consumes much less power as compared to communication and computation subsystems that consume bulk of the available power-budget for a node [116, 30]. It has also been shown that the power required for the communication subsystem even dominates the power required by computation and control subsystem. Pottie et al. [112] showed that assuming a communication channel with Rayleigh fading, the energy cost of transmitting 1 kilo-Byte to a distance of 100 m was approximately the same as that for executing 3 million instructions by a 100 (Million Instructions Per Second) MIPS/Watt processor. And this gap between communication and computation energy is becoming wider due to Moore's law with each newer process technology.

As a result, a lot of efforts are being put to reduce the communication energy of a WSN node. Just to cite a few of them, the use of advanced digital communication techniques (efficient error correction, cooperative MIMO [99]) and network protocols (energy-efficient routing [121] and/or MAC schemes such as S-MAC [145], B-MAC [110], WiseMAC [32] and RICER [84]) have shown to help in improving the energy efficiency for communication (see [4] for a survey).

However, these techniques (e.g. LDPC error correcting codes) may significantly increase the computation workload on the computation subsystem, which in turn (i) impacts the overall energy budget of the system and (ii) may require processing power that would be above the power budget allocated to typical WSN node MCUs.

As a consequence, improving the *computational* energy efficiency of WSN nodes is an important issue. Indeed, we believe that such power and energy savings could open possibilities for more computationally demanding protocols or modulations which, as a result, would provide better quality-of-service (QoS), lower transmission energy and higher network efficiency.

Some of the commercial and academic WSN node platforms existing in the literature are discussed in the following section.

2.4 WSN platforms

There are quite a number of experimental platforms available for WSN research and development. We discuss below a few examples to highlight typical approaches (a detailed overview of current developments can be found, for example, in the work of Hill et al. [55]).

2.4.1 The Mica mote family

Starting in the late 1990s, an entire family of nodes has evolved out of research projects at the University of California at Berkeley, in collaboration with Intel. They are commonly known as the Mica motes, with different versions (Mica, Mica2, Mica2Dot) having been designed [57, 56, 72]. They are commercially available via the company

Crossbow[1] in different versions and different kits. TinyOS [101] is the mostly used OS for these nodes. All these boards feature a microcontroller belonging to the Atmel family, a simple radio modem (usually a TR 1000 from RFM [120]), and various connections to the outside. Sensors are connected to the controller via an I^2C or SPI bus, depending on the version.

The MEDUSA-II nodes [116] share the basic components and are quite similar in design.

2.4.2 BTnodes

The BTnodes [13] have been developed at the ETH Zürich out of several research projects. They feature the Atmel ATmega128L microcontroller, 64B + 180 kB RAM, and 128 kB flash memory. Unlike most of the other sensor nodes, they use Bluetooth technology at radio interface in combination with a Chipcon CC1000 [132].

2.4.3 Telos

The Telos [111] nodes have also been developed at the University of California at Berkeley. They differ in basic components from Mica family. They consist of an MSP430 from Texas Instruments [130] as MCU and a Chipcon CC2420 [131] as an RF transceiver.

2.4.4 PowWow

The PowWow platform [64] developed by our team at INRIA, also uses an MSP430 MCU-core for node control in general and a CC2420 as radio transceiver. It also includes Igloo [1], a low-power FPGA designed by Actel, to configure hardware accelerators for certain compute-intensive applications.

2.4.5 WiseNet

The WiseNet [34] WSN platform has been developed at Swiss Center for Electronics and Microtechnology (CSEM). This power-efficient platform benefits from reduction in energy consumption at the physical layer by using low voltage operations. WiseNet uses a dedicated duty-cycle RF transceiver and a low power MAC protocol (WiseMAC [32]) to lower its communication power consumption. To optimize the startup time and save energy in the RF part, the system invokes different transceiver blocks in a sequence. The lower power baseband blocks are awaken before the radio frequency (RF) circuits. WiseNet node uses an 8-bit CoolRISC [109] core as the general purpose MCU.

2.4.6 ScatterWeb

The ScatterWeb platform [49] was developed at the Computer Systems & Telematics group at the Freie Universität Berlin. This is an entire family of nodes, starting from a relatively standard sensor node (based on MSP 430 microcontroller) and ranges up

[1] http://www.xbow.com

to embedded web servers, which come equipped with a wide range of interconnection possibilities such as Bluetooth, a low-power radio mode, as well as connections following I^2C or CAN (Controller Area Network) protocols are also possible.

Similarly, the Hydrowatch platform is also built on the MSP430F1611-core [39].

Apart from these academic research prototypes, there are already a couple of sensor-node-type devices commercially available, including appropriate housing, certification, and so on. Some of the companies designing commercial WSN nodes include *Millenial*[2] and *Ember*[3].

As our current research work is focused on the power/energy optimization of the computation and control subsystem of a sensor node, in the next section of this chapter we will discuss the evolution of low-power solutions for the control subsystem and present some related work about the low-power microcontrollers and their power optimization techniques.

2.5 Emergence of low-power microcontrollers

Microcontrollers that are used in several wireless sensor node prototypes include the ATMega series by Atmel Corporation or the MSP430 by Texas Instrument. In older prototypes, the Intel StrongArm SA1100 processors have also been used, but it is no longer considered as a practical option (it is included here for the sake of comparison and completeness). These MCUs have several common characteristics such as a simple datapath (8/16-bit wide), a reduced number of instructions, and several power saving modes to have lower power consumption of the system.

The power saving modes adapted in low-power MCUs use VLSI circuit level techniques to reduce the power consumption of the system. Hence, in the following section, we investigate some of the most noticeable power optimization techniques adapted at VLSI circuit level to engineer an energy-efficient design. Some of the key figures about the power consumption of these low-power MCUs are presented afterward.

2.5.1 Power optimization at VLSI circuit level

In the last decade, there have been a large number of research results dealing with power optimization in VLSI circuits (see [28, 108] for surveys on the topic).

Power dissipation in VLSI circuits can be divided into two categories:

- *Dynamic power*: This power dissipation has two sources, (i) capacitance switching (i.e. stage changes) that occurs while the circuit is operating and (ii) short-circuit current passing through the gates.

- *Static power*: It is caused by leakage current between power supply and ground of the circuit.

[2] http://www.millenial.net
[3] http://www.ember.com

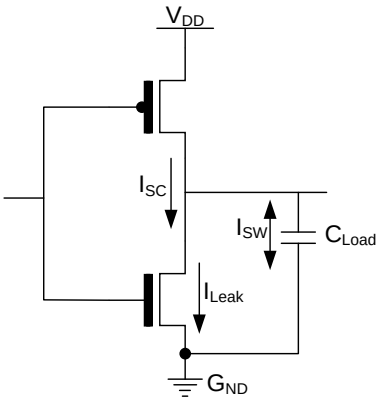


Figure 2.3: Currents contributing to various power consumptions in CMOS circuits.

The total power dissipation of a Complementary Metal-Oxide-Semiconductor (CMOS) gate i is the sum of dynamic power and static power and can be expressed as

$$P_{total} = \frac{1}{2} C_i f_{clk} \alpha_i V_{DD}^2 + V_{DD} I_{sc_i} + V_{DD} I_{leak_i} \tag{2.1}$$

where V_{DD} is the supply voltage, C_i the output capacitance, α_i the activity at the output of gate i, f_{clk} the switching frequency and, I_{sc_i} and I_{leak_i} the short-circuit and leakage current of gate i respectively. Figure 2.3 shows the currents that contribute to the dynamic and static power in a CMOS circuit, I_{SW} is the current that flows while charging and discharging of the output capacitance (stage change), I_{SC} is the short-circuit current flowing when both the NMOS and PMOS transistors conduct for a short duration at input transition whereas I_{Leak} is the leakage current flowing through the gate even when it is not operating.

When a CMOS device is active, its power is usually largely dominated by dynamic power, and becomes roughly proportional to clock frequency. However, in the context of WSN, things are sightly different as the node remains inactive for long periods (MCU duty cycle lower than 1%), and the contribution of static power also becomes significant and can not be ignored.

Moreover, as process technology passes 65 nm and continues toward 45 nm and below, where the operating voltages are lower, and the switching thresholds roll off more rapidly, static power dissipation is expected to exceed dynamic power dissipation and become the dominant contributor to the total power of a device. Therefore, static power minimization must be considered as an integral part of the power reduction strategy. Figure 2.4 points out this ever-increasing share of static power dissipation in overall system power [125].

There are several approaches to reduce the power dissipation in a CMOS circuit, some of them are summarized in the following paragraphs.

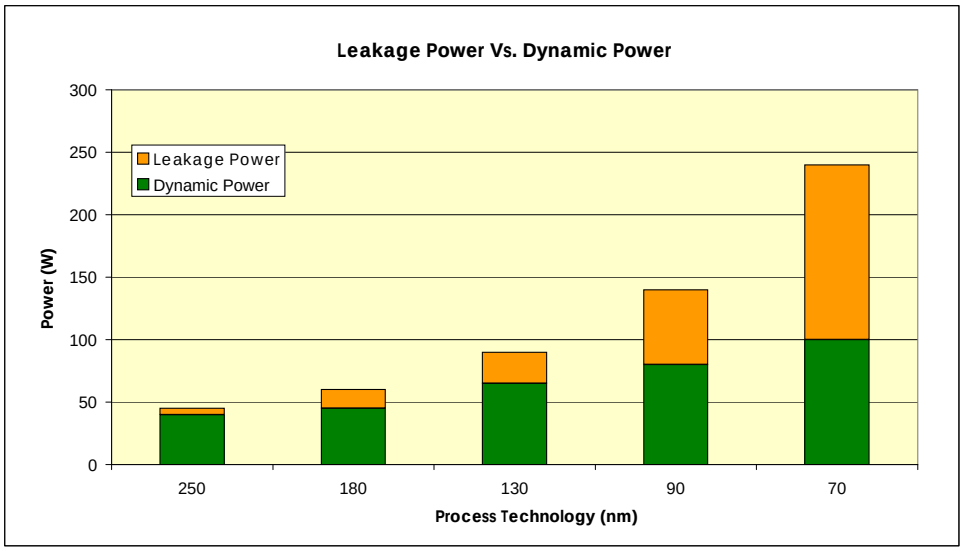


Figure 2.4: Scaling of static and dynamic power consumption with the advancements of process technology.

2.5.1.1 Clock gating

Clock is considered as a major contributor to the power dissipation, as it is the signal with the greatest switching activity. In addition, the clock signal tends to be highly loaded as it has a high fan-out and corresponding high switching capacitance. To distribute the clock and control the clock skew, one needs to construct a clock network (often a "clock-tree") with clock buffers. All of this adds to the capacitance of the clock network. Recent studies (e.g. [143]) indicate that the clock signals in digital systems consume a large percentage (15% to 45%) of the system power. In order to reduce the unnecessary power consumption caused due to clock-trees, *Clock gating* is used.

Clock gating consists in gating the clock signals that drive inactive portions of the circuit. The purpose is to minimize the switching activity on flip-flops and clock distribution lines. For instance, large VLSI circuits such as processors contain register files, arithmetic units and control logic. All the registers in a register file are generally not accessed in each clock cycle. If simple conditions that determine the inaction of particular registers are determined, then power reduction can be obtained by gating the clocks of these registers [19]. When these conditions are satisfied, the switching activity within the register file is reduced to negligible levels.

Figure 2.5 shows one of the typical gated-clock design styles. The "enable-logic" consists of combinational elements and determines whether a clock signal should be supplied to the registers or not. If the output of the enable-logic is equal to zero, clock signal is not propagated to the clock-inputs of the registers. An AND-gate or an OR-gate which satisfies following two conditions is called "gated buffer". (i) The output of the gate is connected to the clock-input of the registers. (ii) One input of

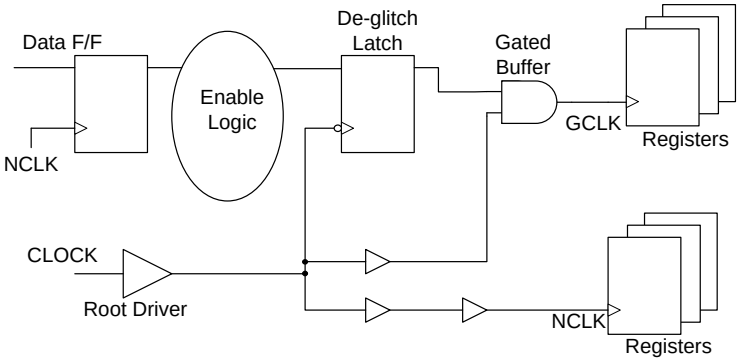


Figure 2.5: Example of gated-clock design.

the gate is an enable signal coming from the "enable-logic", while the other input is the original clock signal. On the contrary, a buffer without gating function is called "normal buffer". The "de-glitch latch" is inserted between the enable-logic and the gated buffer to eliminate glitches which occur in the enable-logic and can result in un-expected behavior of the circuit. A clock signal used in the gated-clock technique is called "gated clock signal" (say GCLK), while the non-gated clock signal is called "normal clock signal" (say NCLK).

Kitahara et al. [75] suggested an automated layout design technique for the clock-gated design and proposed tools to minimize the gated-clock net skew and to maintain timing constraints for enable-logic parts. They proved that about 30% power reduction could be achieved for the whole design by clock-gating using their techniques. Wu et al. [143] used another technique that is based on quaternary representation for behaviors of signals. They presented a method of finding a gated clock signal instead of a normal clock signal using Karnaugh maps, by checking quaternary value of each flipflop in a circuit. Using extended Boolean functions, the authors found the enable logic for clock signals to be connected to the actual circuit and showed that the new clock-gated design reduces the power dissipation by 22%.

Garrett et al. [43] looked at the impact of the physical design on a hierarchical gated clock-tree and its power dissipation. They found that there is an inherent pitfall in implementing gating groups for hierarchical gated-clock distribution because the groups are typically developed at the logic level with no information of the physical layout of the clock-tree. Depending on the distribution of underlying sinks, maintaining gating groups can cause a wiring overhead that is potentially greater than the savings due to reduced switching. Hence, their suggested algorithms took both the logical and physical aspects of the design and generated a more power-optimized solution that results in a 24% more power-saving in clock-trees.

2.5.1.2 Voltage scaling

Voltage scaling is one of the most powerful and frequently-used tools to reduce the dynamic power dissipation. A quadratic improvement can be easily achieved through lowering the supply voltage (Equation 2.1). Although this technique is very effective, the speed of the circuits is degraded as the propagation delay increases with the decrease in supply voltage. Equation 2.2 shows the proportional relation between supply voltage V_{DD} and propagation delay t_{pd}.

$$t_{pd} \propto \frac{V_{DD}}{(V_{DD} - V_{th})^{\alpha}} \tag{2.2}$$

where α is a factor depending on the carrier velocity saturation and is about 1.3 in advanced MOSFETs [67]. A derivative of this approach is to use multiple voltages in a system for different parts of the circuit depending upon their critical path delays. For instance, microprocessors with dual supply voltage scaling technique have the dramatic effect on power consumption reduction. The technique can significantly reduce dissipated power without degrading speed by using lower supply voltage along non-critical delay paths and higher supply voltage along critical delay paths [88]. The main problems of using dual supply voltage scaling in CMOS circuits are the increased leakage current in the high voltage gates when they are driven by low voltage gates, and the routing of two power-supply grids. One solution is to use an additional circuit of level converter, but it introduces area and energy overhead. On the other hand, some researchers proposed Clustered Voltage Scaling (CVS), where no low voltage gate will drive a high voltage gate [139]. However, both of these techniques introduce additional constraints to the dual supply voltage scaling process, and reduce the overall energy savings.

In contrast to the above-mentioned techniques that implement the voltage scaling at circuit level, there have been several other approaches that proposed to implement multiple supply voltages at other levels of design-cycle as well. For instance, Chang et al. [20] presented a dynamic programming technique for solving the multiple supply voltage scheduling problem in both non-pipelined and functionally pipelined datapaths. The scheduling problem was concerned to the assignment of a supply voltage level (selected from a fixed and known number of voltage levels) to each operation in a "Data Flow Graph (DFG)" so as to minimize the average energy consumption for given computation-time or throughput constraint or both of them. They showed that using three supply voltage levels on a number of standard benchmarks, an average energy saving of up to 40% can be obtained compared to using a single supply voltage level.

2.5.1.3 Transistor sizing

Transistor sizing in a combinational circuit significantly impacts the circuit delay and power dissipation. If the transistors in a given gate are increased in size, the delay of the gate decreases, however, it increases the delay and the load capacitance of the *fan-in* gates. A typical approach to find the optimum transistor size, given a delay constraint, is to compute the slack at each gate in the circuit, where the slack of a

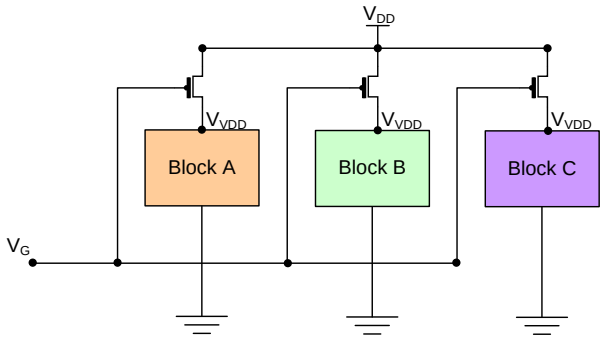


Figure 2.6: The use power gating to reduce the overall circuit power.

gate corresponds to how much the gate can be slowed down without affecting the critical delay of the circuit. Sub-circuits with slacks greater than zero are processed for transistor-size reduction until the slack becomes zero, or the transistors reach the minimum size. A similar approach is used in the work by Tan et al. [127]. However, for a given delay-constraint finding an appropriate sizing of transistors that minimizes power dissipation is a computationally difficult problem.

2.5.1.4 Power gating

One important issue with all of the above-mentioned techniques so-far is that they work for *dynamic power* reduction of a circuit and are poorly suited to WSN nodes as they significantly increase the total silicon area by adding extra components (e.g. a level converter in case of dual voltage scaling, gated buffer and enable-logic for clock gating), and therefore have a negative impact on *static power* dissipation.

However, one exception is *power gating*, that consists in turning-off the power supply of inactive circuit components [12, 87] which helps reducing both dynamic and static power. It is thus a very efficient optimization for devices in which components remain idle for long time periods.

The technique consists in adding a *sleep transistor* between the actual V_{DD} (power supply) rail and the component's V_{DD}, thus creating a *virtual supply voltage* called V_{VDD} as illustrated in Figure 2.6. Similarly a *sleep transistor* between the actual GND (ground) rail and the component's GND can also be added, creating a *virtual ground* called V_{GND}. The sleep transistor, in first case, allows the supply voltage of the block to be cut off to dramatically reduce leakage currents. In practice, Dual V_T CMOS or Multi-Threshold CMOS (MTCMOS) techniques are used for power gating implementation [95, 69]. Research work is also being done on the sleep transistor sizing to further reduce the leakage power caused by the sleep transistor insertion [22].

Since one centralized sleep transistor design suffers from large interconnect resistances between distant blocks, such resistance has to be compensated by extra large sleep transistor area that could result in extra load capacitance and delay for driving logic. Hence, two approaches have been proposed to divide the overall circuit into seg-

ments and apply several sleep transistors to achieve a more-efficient design. Anis et al [7] proposed a cluster-based design structure, where each cluster, consisting of several gates, is accommodated by a sleep transistor separately. The size of the sleep transistor is determined by the current of the cluster. Their approach achieved a 90% reduction in static power consumption as well as 15% reduction in dynamic power consumption. Long et al. [87] proposed another approach that uses a distributed network of sleep transistors. This approach is better than the cluster-based approach in terms of sleep transistor area and circuit performance and obtains sleep transistor networks that are 70% more area-efficient than the cluster-based networks.

Power gating has already been used in the context of high performance CPUs [59], and FSM implementations [113] where parts of the design are switched on/off according to their activity. The approach helps in reducing the static power dissipation for FPUs of a high-end CPU by up to 28% at the price of a performance loss of 2%, for FSMs the average reported power reduction was also 28%. In the context of WSN, where nodes remain idle most of the time, such a technique has obvious advantages, and is therefore intensively used for implementing the low power modes of typical node MCUs.

Most of the above-mentioned VLSI power-optimization techniques have been deployed in ultra low-power microcontrollers targeted toward low-power embedded system applications. A brief overview of these Commercially-Off-The-Shelf (COTS) microcontrollers that are currently being used in most of the existing WSN nodes is presented in the following section.

2.5.2 Commercial low-power MCUs

In recent years, due to rapid evolution of embedded systems, several low-power microcontrollers have evolved. However, since these MCUs are designed for low-power operation across a wide range of embedded system application settings, they might not be very well-suited to the event-driven behavior of WSN nodes as they are based on a general purpose, monolithic compute engine. In this section we present the features of some of the most notable COTS microcontrollers being used by the WSN community for WSN node development.

Texas Instruments MSP430 series

The MSP430 family features a wider range of low-power 16-bit MCUs. For instance, the MSP430F1611 requires 500 μA (@ 1 MHz and 3.0V) whereas an updated series, designed for WSN nodes, the MSP430F21x2 consumes approximately 250 μA (@ 1 MHz and 2.2V) [129]. There are four sleep modes in total. The deepest sleep mode, LPM4, consumes only 0.1 μW, but the controller can only be woken up by external interrupts in this mode. The MSP430 has a maximum operating frequency of 16 MHz.

Atmel ATMega series

Atmel Corporation has also developed a wide range of 8-bit MCUs that are used by a variety of embedded system applications. In particular, for WSN node platforms,

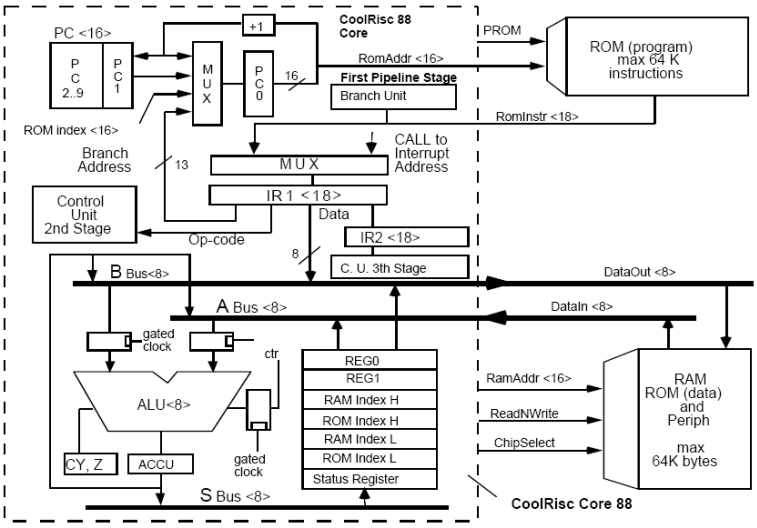


Figure 2.7: Architecture of CoolRISC 88 processor (extracted from the work of Piguet et al. [109]).

ATmega128L and ATmega103L have been used [9, 8]. ATmega128L consumes an average 8 mA (@ 8 MHz and 3.0V) whereas the older version ATmega103L, with an operating frequency of 4 MHz max., consumes approximately 5.5 mA (@ 4 MHz and 3.0V). The ATmega128L has a maximum operating frequency of 8 MHz.

EM Microelectronic CoolRISC

EM Microelectronic has also come-up with an ultra low-power solution for 8-bit microcontroller called CoolRISC [33]. An EM6812 (based on CoolRISC) consumes approximately 120 μA (@ 1 MHz and 3.0V) [33], whereas it has a maximum operating frequency of 5 MHz (2.5 MIPS).

Historically, original CoolRISC was a low-power processor designed to achieve a lower value of *Clock per Instruction (CPI)* to reduce the power consumption by working in lower frequency range. At that time, CoolRISC 88 achieved a $CPI = 1$ by providing 10 MIPS performance and an operating frequency in the range of 1 to 10 MHz [109]. The legacy CoolRISC 88 having a register file of 8 registers is shown in Figure 2.7 (extracted from the work by Piguet et al. [109]). It was built in 1 μm process technology that resulted in a lower power consumption. It consumed around 60 μA at 3.0 V while working at 1 MHz.

NXP LPC111x

NXP Semiconductors have recently launched a series of MCUs, LPC111x that are based on ARM Cortex-M0 [100]. These are 32-bit MCU cores with an average of 3.0 mA

WSN MCU	Normalized Power	Actual Power
ATmega103L	66 mW (@ 16 MHz)	5.5 mA (@ 4 MHz, 3.0V)
ATmega128L	48 mW (@ 16 MHz)	8 mA (@ 8 MHz, 3.0V)
MSP430F1611	24 mW (@ 16 MHz)	500 μA (@ 1 MHz, 3.0V)
MSP430F21x2	8.8 mW (@ 16 MHz)	250 μA (@ 1 MHz, 2.2V)
EM-6812 (CoolRISC)	5.76 mW (@ 16 MHz)	120 μA (@ 1 MHz, 3.0V)
LPC111x (ARM Cortex-M0)	13.2 mW (@ 16 MHz)	3.0 mA (@ 12 MHz, 3.3V)
StrongArm SA-1100	40 mW (@ 16 MHz)	400 mA (@ 160 MHz, 2.0V)

Table 2.2: Actual and normalized power consumption for various low-power MCUs.

(@ 12 MHz, 3.3V) that can run up to an operating frequency of 50 MHz. LPC111x consumes around 6 μA (@ 3.3V) in deep-sleep mode without any active clock.

Intel StrongArm SA-1100

The Intel StrongARM [66] provides three power modes: (i) In normal mode, all parts of the processor are fully powered. Power consumption is up to 400 mW (@ 160 MHz, 2.0V). (ii) In idle mode, clocks to the CPU are stopped while clocks to peripherals are active. Any interrupt will cause return to normal mode. The power consumption is up to 100 mW. (iii) In sleep mode, only the real-time clock remains active. Wake-up occurs after a timer interrupt and takes up to 160 ms and the power consumption is up to 50 μW.

Table 2.2 summarizes power consumptions of all the above MCUs at a normalized operating frequency of 16 MHz.

2.5.3 WSN-specific sub-threshold controllers

Apart from the general-purpose COTS processors, there are several WSN-specific controller implementations that have been proposed by the research community. These controllers try to exploit the WSN-specific characteristics such as *event-centric behavior* and *asynchronous communication* to achieve an extremely low-power consumption as well as a considerably lower energy per instruction. In this section, we will briefly discuss some of the most significant WSN-specific processors present in the literature.

2.5.3.1 SNAP/LE processor

The SNAP/LE, proposed by Ekanayake et al. is an ultra low-power asynchronous processor [31]. The processor instruction set is optimized for WSN applications, with support for event scheduling, pseudo-random number generation, bit-field operations, and radio/sensor interfaces. SNAP/LE has a hardware event queue and event coprocessors, which allow the processor to avoid the overhead of operating system software (such as task schedulers and external interrupt servicing), while still providing a straightforward programming interface to the designer. Figure 2.8 shows the basic micro-architecture of SNAP/LE. The timer coprocessor is used for timing synchronization and generates

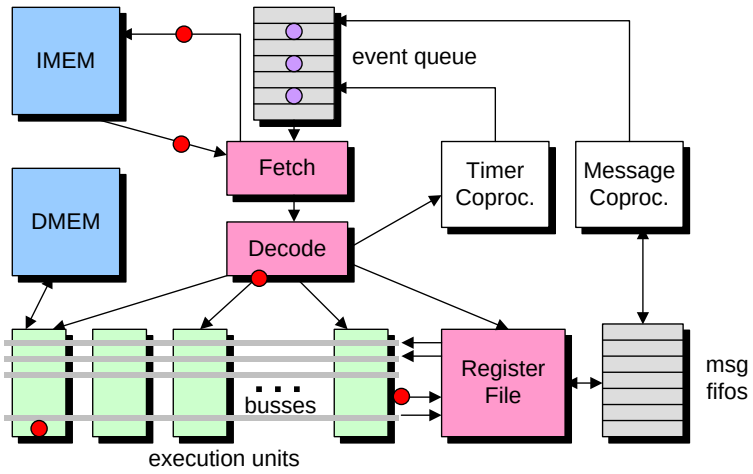


Figure 2.8: Microarchitecture of the SNAP/LE processor showing major components.

internal time-outs whereas message coprocessor is used for message exchange with the RF-transceiver. The decode stage in Figure 2.8 is a normal stage whereas instruction-fetch along with the event queue forms a FIFO-based hardware task scheduler. The instruction-fetch waits for an event in the event queue and uses this event as an index to access the SNAP/LE's "event handler table" to address a proper event handler. It then starts fetching the instructions concerning the relative event handler until a "done" instruction is encountered. The instruction-fetch then checks the event queue for a possible next event. The dots (in the figure) correspond to the instruction tokens being processed by the SNAP/LE processor core.

Running at its highest core voltage of 1.8 V, the SNAP/LE core consumes under 300 pJ/instruction while working at 600 mV it consumes 75 pJ/instruction with several instructions as low as 25 pJ/instruction. The SNAP/LE processor is implemented in 180 nm process technology.

2.5.3.2 Accelerator-based WSN processor

Hempstead et al. [53] proposed a processor design that is compliant with the accelerator-based computing paradigm, including hardware accelerators for the network layer (routing) and application layer (data filtering). Moreover, the architecture can disable these accelerators via V_{DD}-gating to minimize leakage current during the long idle periods common to WSN applications. They implemented a system architecture for WSN nodes in 130 nm CMOS technology that operates at 550 mV and 12.5 MHz.

Figure 2.9 shows the general architecture of this processor. There is an event-handler implemented in hardware that performs the regular event management, then there is a general purpose MCU for irregular event management and finally there are several accelerators for application and network level tasks such as message routing and data filtering etc. Since the commonly used metric of energy-per-instruction can not be easily

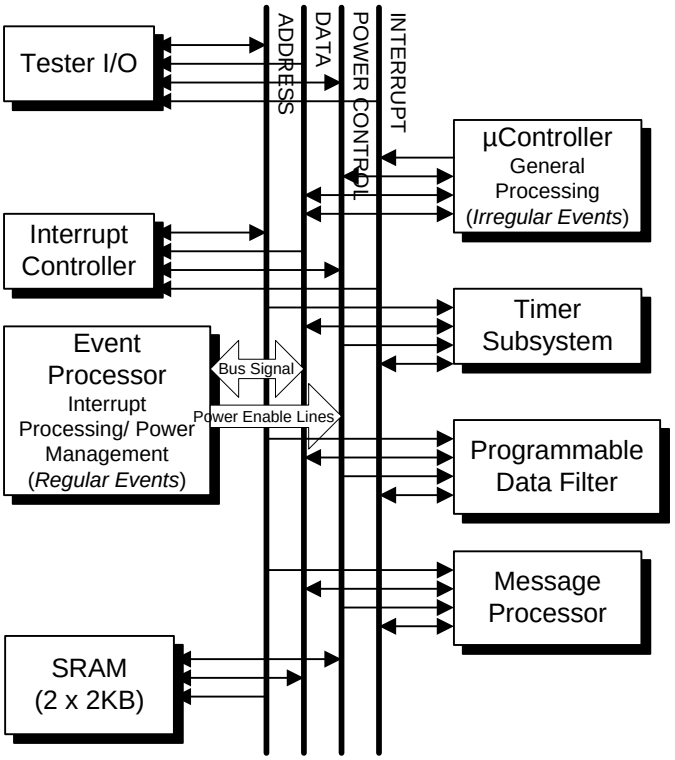


Figure 2.9: System architecture of the accelerator-based WSN processor.

applied to accelerator-based systems, the authors introduced the concept of "energy-per-task". They defined a task as a collection of dependent computations that are executed periodically. They presented measurements of a task similar in nature to a volcano monitoring application. This task took 131 cycles to execute and consumed 678.9 pJ at 550 mV and 12.5 MHz. An equivalent routine written for the Mica2 mote required 1532 instructions. Using this information, they computed the energy per equivalent instruction as 0.44 pJ/instruction for accelerator-based mode while 3.4 pJ/instruction in general purpose mode for their processor.

Even though their idea of benefiting from hardware acceleration and power-gating is similar to ours, they design all of their accelerators manually and work in subthreshold voltage domain that is prone to inconsistencies due to thermal and process variations. On the other hand, we propose a complete automated design-flow for the generation of specialized hardware blocks and our designs work at above-threshold voltage domain to avoid complications. Moreover, the authors did not provide any details about the general purpose microcontroller present in their system and its Instruction-Set Architecture (ISA), hence it is difficult to judge their claims about an energy efficiency of 3.4 pJ/instruction that is 20x times better than an asynchronous processor (SNAP/LE) performing nearly the same job.

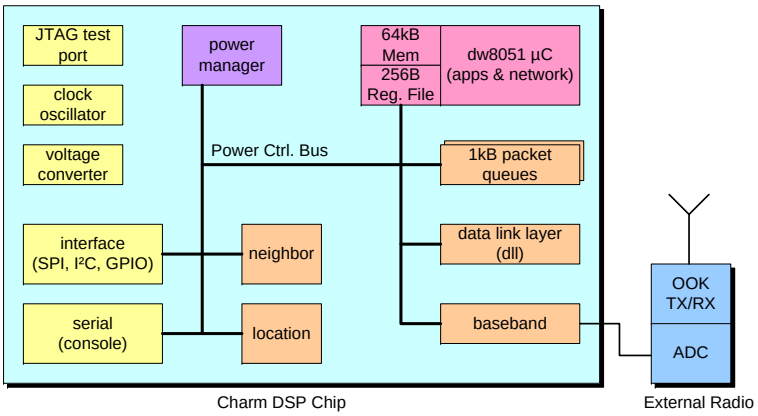


Figure 2.10: Block diagram of the Charm protocol processor.

2.5.3.3 Charm processor

The Charm processor [123] developed at the University of California at Berkeley is a protocol processor implemented in 130 nm technology and implements a protocol stack tailored for WSN applications. Its subsystems follow the OSI model and include the application, network, data link, and digital baseband portion of the physical layers.

As shown in Figure 2.10, the processor chip contains a synthesized 8051-compatible microcontroller with 64 kilo-Byte of program/data RAM, two 1 kilo-Byte packet queues, a custom data-link layer (DLL), a neighborhood management subsystem, digital portion of a custom baseband, a location computation subsystem, and several external interfaces. Charm processor uses a unique approach that, instead of using V_{DD}-gating, reduces the power rail voltage to a "Data Retention Voltage (DRV)" that maintains the state in the logic, while still reducing leakage current. Hence, the system has a trade-off of data retention for a relatively higher leakage power dissipation. It works at 1.03 V and consumes 96 pJ/instruction [53].

2.5.3.4 Phoenix processor

Seaok et al. developed the Phoenix processor (shown in Figure 2.11), that works at 500 mV and consumes as low as 2.8 pJ/cycle [122]. The Phoenix processor exploits both the voltage scaling as well as a comprehensive sleep transistor technology to obtain an ultra low-power processor design. The CPU works on an even-driven paradigm and the authors claim to intentionally use an older low-leakage 0.18 μm process technology to further reduce the static power consumption.

However, the authors did not provide any information about the CPU being used or its ISA. Similarly, no information about the clock-cycles taken by the CPU instructions is provided. Hence, it impossible to interpret their energy efficiency metric "energy/cycle" to normal metric such as "energy/instruction".

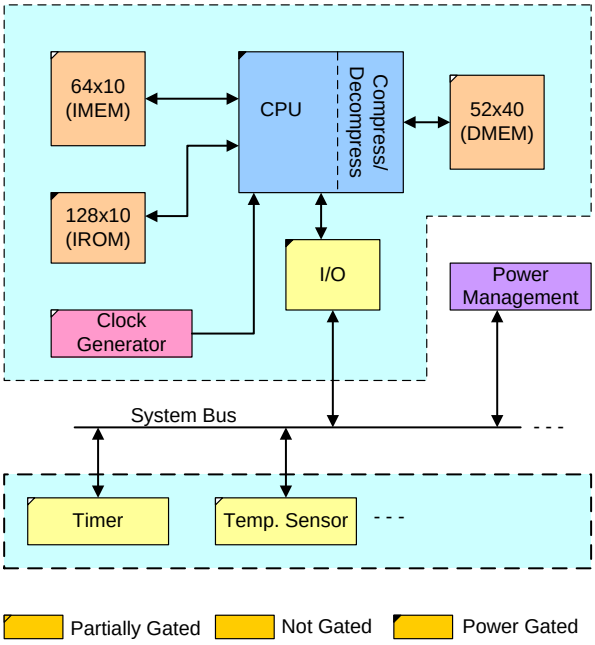


Figure 2.11: Block diagram of the Phoenix processor.

2.5.3.5 BlueDot

Very recently, Raval et al. [119] have proposed BlueDot, a low-power TinyOS-tuned processor platform for WSN nodes implemented in 130 nm process technology. The BlueDot is compatible with Atmel ATmega128L series processors and an application code written for ATmega128L-centric nodes can run on BlueDot-based nodes as well. A BlueDot-based platform can also benefit from several hardware-accelerators that can be used to implement frequently used application/control codes such as communication with RF-transceiver through SPI-bus. The BlueDot platform, with an optimized processor (having WSN-specific Instruction-set Architecture (ISA)) and RF-accelerator, consumes 26 pJ/instruction on average. It consumes around 48% less energy than the baseline ATmega128L-equivalent processor when executing the same WSN application suite that is around 1.5 mJ on average. However, the authors did not provide the area and power usage comparison w.r.t. the base-line ATmega128L processor. It is quite possible that the overall area of the node as well as static power is increased by adding the hardware accelerators along with the base-line processor that could result in a less power-efficient design in standby mode.

2.5.4 Conclusion

Though all of these WSN-specific processors show impressive energy efficiency in terms of *Joules/instruction* as compared to general purpose COTS processors mentioned ear-

lier, they also suffer from their inherent weaknesses. For instance, subthreshold logic is highly susceptible to temperature and process variations. In addition, due to the low voltage swing, noise arising from other on-chip components could be an issue for commercial WSN applications. As far as asynchronous processors are concerned, it can be difficult to integrate asynchronous logic into conventional, commercial synchronous design flows for low-cost System-on-Chip (SoC) solutions. For this reason, asynchronous logic is considered unattractive for many applications. Moreover, most of these processors are manually designed and optimized and no automatic design and programming tool exists for them.

Due to these inherent problems of subthreshold and asynchronous WSN-specific processors and a very high power consumption of general purpose COTS processors, we focused on the development of a novel approach that is based on hardware specialization and power gating that is built upon a hybrid concept of High-Level Synthesis (HLS) and retargetable compilation for ASIP (Application Specific Instruction-Set Processor). The next chapter presents a comprehensive state-of-the-art about HLS and ASIP design tools.

Chapter 3

High-level synthesis and application specific processor design

As discussed in Section 1.3.3, our work consists of two distinct design-flows. The first one takes the overall system-level behavioral description of application (using a Domain Specific Language (DSL)) as an input and generates an RTL VHDL description of the global system monitor. On the other hand, the second design-flow takes the C-specification of the tasks present in the application task graph and generates the specialized hardware blocks, called hardware micro-tasks, for each of them. As our design-flow for hardware micro-task generation is a hybrid of High-Level Synthesis (HLS) and Application Specific Instruction-set Processor (ASIP) design methodologies, we present, in this chapter, the existing work done in the domain of HLS and ASIP design. The chapter starts with the generic methodology for HLS and discusses the existing algorithms as well as academic and commercial tools available for HLS. In a similar fashion, it then presents the generic methodologies for ASIP design and corresponding algorithms. The chapter finally concludes with a discussion on some of the existing academic and commercial tools for ASIP design.

3.1 High-Level Synthesis (HLS)

Rapid advancements in silicon technology and the increasing complexity of applications in recent decades have forced design methodologies and tools to move up to higher abstraction levels. Raising the abstraction levels and accelerating automation of both synthesis and verification tools have been major key factors in the evolution of design process. Moreover, the use of high-level models has enabled system-designers, rather than circuit-designers, to be productive and to match the industry trends which is delivering an increasingly large number of integrated systems in place of integrated circuits.

The HLS design relates to leaving the implementation details to the design algo-

rithms and tools, and thus represents an ambitious attempt by the research community to provide capabilities for "algorithms to gates" for a period of almost three decades [48]. HLS takes as its input a behavioral description in the form of a high-level language and generates an RTL circuit [85]. In the next section, we summarize the generic design-flow for HLS found in the literature.

3.1.1 Generic HLS design-flow

Coussy et al. [24] presented a generic design-flow that is shared by most of the HLS tools. This design-flow consists of seven major steps that are highlighted in Figure 3.1. We briefly discuss some of these design steps:

- ***Compilation***: HLS always begins with the compilation of the functional specification. This first step transforms the input description into a formal representation. This step includes several code optimizations such as dead-code elimination, constant folding and loop transformations etc. and generates an intermediate specification that can be in the form of a CDFG.

- ***Allocation***: Allocation defines the type and the number of hardware resources (e.g. functional units, storage, or connectivity components) needed to satisfy the design constraints. These components are selected from the RTL component library. The library must also include component characteristics (such as area, delay, and power) and its metrics to be used by other synthesis tasks.

- ***Scheduling***: All operations required in the specification model must be scheduled into cycles. Depending on the functional component to which the operation is mapped, the operation can be scheduled within one (or several) clock cycle(s). The operations can be scheduled in parallel or sequential fashion depending upon the data dependencies between them and if there are sufficient resources available at a given time.

- ***Binding***: This step binds the variables carrying a life-time of several clock cycles to storage resources. Similarly, the operations are bound to the functional units that are optimally selected for given design constraints.

- ***Generation***: Once decisions have been made in the preceding tasks of allocation, scheduling, and binding, the goal of the RTL architecture generation step is to apply all the design decisions made and generate an RTL model of the synthesized design.

Now we briefly discuss the algorithms that have been mostly used in the existing work during different steps of the HLS design-flow such as operation scheduling, resource allocation and binding.

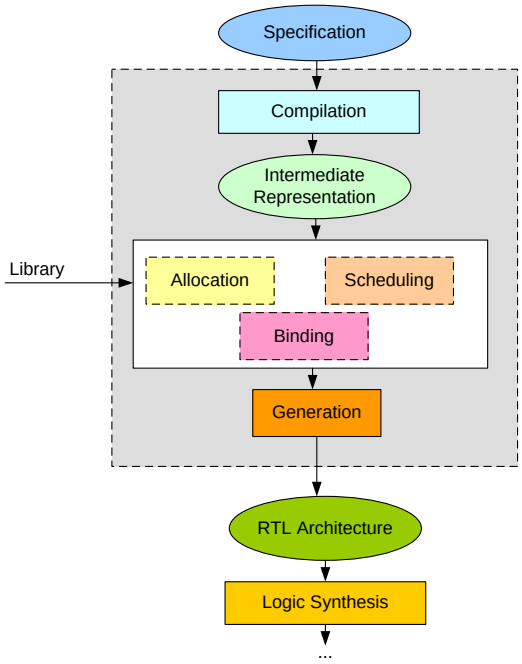


Figure 3.1: Design methodology for high level synthesis (HLS)

3.1.2 Scheduling

Operation scheduling consists of assigning each operation at behavioral level (e.g. DFG) to a control step (also called a C-step [106]). Generally speaking, scheduling determines the cost-speed trade-off of a hardware design generated through HLS. If the target design is subject to a timing/speed constraint, the scheduling algorithms will try to meet the timing constraint through operation parallelization and an optimum algorithm will provide a solution with the minimum hardware resources under the given timing constraint. On the other hand, if there is a limit on the cost (area or resources) of the target design, the scheduling algorithms will serialize the operations to meet the resource constraint. In this case, the optimum solution will be the one that reduces the overall execution time while keeping the resource constraint into consideration.

Scheduling, without resource constraints is simple as it consists in a topological sorting of the operations and can be solved in polynomial time for an optimal solution. Two of the algorithms to solve such scheduling are discussed in the following paragraphs.

3.1.2.1 ASAP scheduling

The simplest scheduling technique is As Soon As Possible (ASAP) scheduling where the operations in the DFG are scheduled step-by-step from the first control step to the last. An operation is called ready operation if all of its predecessors are scheduled. This procedure repeatedly schedules ready operations to the next control step until all the

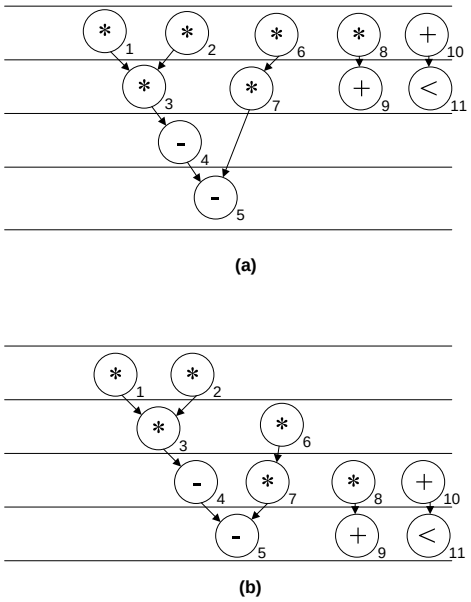


Figure 3.2: (a) ASAP scheduling (b) ALAP scheduling

operations are scheduled.

3.1.2.2 ALAP scheduling

As Late As Possible (ALAP) scheduling has a very similar approach to ASAP scheduling. In contrast to ASAP, ALAP scheduling schedules the operations from the last control step toward the first performing a backward covering. An operation is scheduled to the next control step if all of its successors are scheduled. Figure 3.2 shows an example of ASAP and ALAP scheduling.

3.1.3 Resource-constrained scheduling

Scheduling, under resource constraints (whether computation resources or computation time) is a computationally difficult problem that lies in the domain of NP-Complete problems [15]. Some of the mostly used scheduling algorithms and their implementations are discussed in the following paragraphs.

3.1.3.1 List scheduling:

One commonly used approach is called "list scheduling" or "resource-constrained scheduling" in which we specify a hardware constraint and use an algorithm to minimize the total execution time that satisfies the given constraint. Like ASAP, the operations in the DFG are assigned to control steps from the first control step to the last. The ready operations are given a priority according to heuristic rules and are scheduled into the

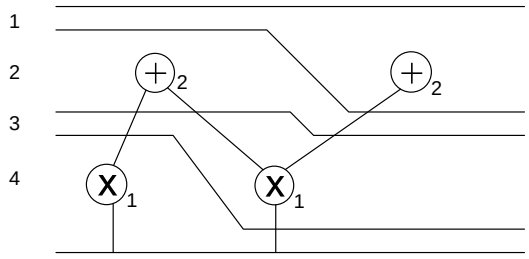


Figure 3.3: List scheduling with deferred operations.

next control step according to this predefined priority. When the number of scheduled operations exceeds the number of resources, the remaining operations are deferred to next control step. Which operations to defer often depends on some local priority such as urgency. In Figure 3.3, for example, two add operations may be scheduled in the first control step, so we must defer one of them. Since they are both on the critical path, they have the same urgency, so we could choose either one. In the figure, the left addition is deferred to the second control step.

3.1.3.2 Force-Directed Scheduling (FDS):

The force-directed scheduling (FDS), also called "time-constrained scheduling", reduces the number of functional units, registers, and busses required. The strategy is to place similar operations in different control steps so as to balance the concurrency of the operations assigned to the units without increasing the total execution time. In FDS, *force* values are calculated for all operations at all feasible control steps. The pairing of operation and control step that has the most attractive force is selected and assigned. After the assignment, the forces of the unscheduled operations are re-evaluated. Assignment and evaluation are iterated until all the operations are assigned.

3.1.3.3 Force-Directed List Scheduling (FDLS):

Paulin et el. [106] suggested a hybrid of both of the above-mentioned approaches that is called "force-directed list scheduling (FDLS)". It provides a solution considering both timing and hardware resource constraints. Similar to list scheduling, FDLS also schedules the operations in the DFG to different control steps from the first control step to the last. However, the local priorities of operations that are used to defer them to next control step are defined by the *force* function used in FDS, instead of the usual metrics such as urgency.

The complexity of both FDS and FDLS is $O(n^2)$ where n is number of operations present in a CDFG. However, all the above mentioned algorithms i.e. list scheduling, FDS and FDLS are heuristic algorithms that provide an approximate solution while the optimality of the solution cannot be guaranteed.

3.1.3.4 Mixed Integer Linear Programming (MILP)-based approach:

Instead of using heuristic algorithms, Hwang et al. [61] proposed an MILP-based formal approach to solve the problem of scheduling for HLS. They proposed solutions for three different types of scheduling. (i) time-constrained scheduling (ii) resource-constrained scheduling and (iii) feasible-constrained scheduling, that takes both time and resource constraints and outputs a solution if it exists. The authors used a combination of ASAP, ALAP, list scheduling and MILP formulation to obtain a relatively less-complex solution with a complexity $O(s.n)$ where s and n are number of control steps and number of operations respectively.

3.1.4 Resource allocation/binding

The resource binding assigns the operations and memory accesses within each clock cycle to available hardware units. A resource such as a functional, storage, or interconnection unit can be shared by different operations, data accesses, or data transfers if they are mutually exclusive. For instance, two operations assigned to two different control steps are mutually exclusive since they will never execute simultaneously and as a result, they can be bound to the same functional unit. Binding consists of three sub-tasks based on the underlying unit type:

- ***Storage binding*** assigns variables to storage units. Storage units can be of many types including registers, register files, and memory units.
- ***Functional-unit binding*** assigns each operation in a control step to a functional unit. A functional unit can execute only one operation per clock cycle.
- ***Interconnection binding*** assigns an interconnection unit such as a multiplexer or bus for each data transfer among ports, functional units, and storage units.

There exist several approaches to resolve the problem of resource binding. Some approaches (e.g. [137]) solve all three sub-tasks simultaneously whereas other approaches either solve them one-by-one (e.g. Hybrid ALlocation (HAL) [107]) or two at a time (e.g. [102]). There are two classes of resource allocation and binding problem that are further discussed in the following section.

3.1.4.1 Interval-graph based allocation

The first category of the resource allocation problems is simpler of the two where the liveness of the variables can be represented an *interval-graph* and an optimum solution can be found in polynomial time with a single traversal of the *interval-graph*. One example of such algorithms is called "Left-Edge Algorithm (LEA)" that is discussed below.

Left-Edge Algorithm (LEA): LEA is an algorithm used for register allocation. It has been used previously for track assignment in channel routing [78]. It has been proven optimal, and is of complexity $O(n^2)$ where n is the number of variables present in a CDFG. Essentially, the track assignment problem is solved as follows:

1. Sort the wire segments in increasing order of their left edges.
2. Assign the first segment (the leftmost edge) to the first track.
3. Find the first wire whose left edge is to the right of the last selected wire and assign it to the current track.
4. If no more, wires can be assigned to the current track, start a new track and begin again from Step 2.
5. Repeat until all wires are assigned to tracks.

The life-time of each variable is mapped into a net interval in a channel routing problem. The number of tracks needed by the LEA is equal to the number of registers allocated and variables whose intervals are in the same track are assigned to the same register.

3.1.4.2 Conflict-graph based allocation

The second category of resource allocation problems is harder in complexity. In such allocation problems, the liveness of the variables is difficult to calculate and it must be represented by an *interference-graph* or a *conflict-graph*. This kind of allocation problems are computationally hard and lie in the domain of NP-Complete problems. The most common algorithms to solve such resource allocation and binding problems are heuristic algorithms. Two of such algorithms are discussed in the following paragraphs.

Heuristic *clique* partitioning [137]: For register allocation, this algorithm builds a graph where each vertex represents a variable and an edge exists between two vertices if, and only if, the two corresponding variables can share a same register (i.e. they have mutually-exclusive life-times). The graph is then partitioned into a number of *cliques* (a *clique* is a *complete* subgraph such that there exists an edge between every vertex of the subgraph). The number of *cliques* partitioned is the number of registers needed and a register is allocated for those variables corresponding to the vertices in each *clique*. For functional unit allocation, the vertex in the constructed graph represent the operations and an edge between the two vertices exists if, and only if, the two corresponding operations will not be performed in a same control step. The interconnect binding is performed in the similar fashion.

Graph coloring algorithm: Graph coloring algorithm is an assignment of labels, traditionally called *colors*, to elements of a graph subject to certain constraints. Formally speaking, the situation is described by a graph $G = (V, E)$ with vertex-set V and edge-set E formed by all pairs of incompatible elements. Partitioning of V into k subsets is equivalent to coloring the vertices of G with k colors.

In simpler form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a vertex coloring. In register allocation, an interference graph is constructed, where vertices are symbolic registers and an edge connects two nodes if they are needed at the same time (i.e. they are alive at the same time and cannot be assigned to the same physical register). If the graph can be colored with k colors then the variables can be stored in k physical registers. Both graph coloring and register allocation (except for some special cases, such as expression trees) are NP-Complete problems. Fortunately, there exist several methods such as greedy algorithm, breadth-first search, brute-force search to solve the problem of vertex coloring with a linear-time approximation and they could yield to good results but without ensuring optimality.

After discussing the existing design methodology and corresponding algorithms for each stage of the HLS design-flow, we present some academic as well as commercial tools that are based on these principles and are being used by the HLS research community.

3.2 Power-aware HLS tools

HLS tools have been developed for targeting different design constraints such as low-power, low silicon foot-print, FPGA resource optimization, wordlength optimization for Digital Signal Processing (DSP) applications etc. Since our work targets a lower power consumption of the generated hardware, we will mainly focus on the power-aware HLS tools.

3.2.1 SCALP

Ragunathan et al. [117] proposed SCALP, an iterative-improvement-based low-power datapath synthesis tool that targets data-oriented application domain. It adopts a heuristic based approach for solving different HLS tasks such as scheduling, resource binding and resource sharing etc. It also offers various optimizations such as compiler-level transformations, appropriate clock and supply-voltage selection for a power-efficient design, retiming, datapath merging as well as slower functional-unit selection (if permitted by the timing constraint) to avoid excessive switching capacitance and reduce the dynamic power of the resultant hardware. One of the main problems with SCALP, in our point of view, is that it is targeted toward data-dominated applications and its feasibility toward control-oriented applications (such as WSN) is not shown. It also generates an interconnect-unaware solution that does not consider the power dissipated by the interconnects and thus provides a relatively less power-efficient solution.

3.2.2 Interconnect-Aware Power Optimized (IAPO) approach

Zhong et al. [85] provided an improvement over some HLS tools by taking into consideration of the power consumed by interconnect. They added a notion of interconnect-aware resource-binding in SCALP. They considered multiplexer-based interconnects in their system and provided two methodologies for improved interconnect-bindings:

- ***Neighborhood-aware binding***: the nodes in a CDFG that exchange data with each other are mapped to the datapath units (DPUs) such that they are physically placed closer to each other during floor-planning to reduce the communication cost.
- ***Communication-sensitive binding***: the authors added a weighted communication gain to the cost gain for DPU-sharing moves. It is based on the unit-length switched capacitance of the data exchange between the corresponding two DPUs. This tends to merge DPUs, which have intensive data-exchange between them.

The generated hardware through IAPO approach consumes 53% less power in interconnects while 26% less overall power as compared to interconnect-unaware design generated by SCALP.

3.2.3 LOPASS

The number of input ports of the multiplexers present in a circuit can greatly affect the size of the FPGA resources needed to implement that circuit. Resultantly, this could lead to a higher power consumption. Chen et al. [27] proposed LOPASS, a low-power HLS tool that is suitable for FPGA architectures and targets data-dominated applications also. It uses FDS for early scheduling and simulated annealing engine for resource allocation and binding. It uses function unit optimizations such as *merge*, *split*, *reselect*, *swap* and *mix* (at random) to generate new solutions. After each step of optimization, list scheduling is used to check the overall timing and latency constraints. Since a wide-port multiplexer can be quite expensive in terms of resources on FPGA, a weighted bipartite matching algorithm is used to minimize the number of ports in multiplexers used by the design to finalize the LOPASS flow. The authors showed that LOPASS is able to reduce the power consumption of a design by 35% when compared to the results of *Synopsys Behavioral Compiler*.

3.2.4 HLS-pg

HLS for power-gated circuits is a relatively new domain in HLS tools. The overhead of the state-retention storage required to preserve the circuit state in standby mode is an inherent problem in designing power-gated circuits. Retention storage size reduction is known to be the key factor in minimizing the loss of power-saving (achieved by power-gating). The bigger is the size of state-retention storage, the lesser will be the overall benefits of the applied power-gating. Choi et al. [23] targeted a new issue in HLS and proposed an approach that addresses the minimization of the retention storage size

present in a design. They targeted scheduling and resource allocation problem where the scheduling problem targets the minimization of the number of state-retention registers and is solved with an MILP formulation. Register allocation problem is solved using a heuristic vertex coloring algorithm. They showed that for 65 nm CMOS technology, HLS-pg generates circuits with 27% less leakage current and with 6% less circuit area and wire-length, compared to the power-gated circuits produced by conventional HLS.

The HLS-pg approach though targets the power-gated specialized hardware synthesis, it is inherently different than our approach as we do not target the HLS of power-gated circuits. Instead we use our design-flow to generate non-power-gated hardware micro-tasks and the power-gating is added to our design (at transistor-level) after its generation. Moreover, the benchmarks used by the authors for their experiments (such as IIR7, ELLIPTIC and WAVELET) show that their design-flow is targeted toward data-intensive applications.

3.3 HLS tools targeting other design constraints

Though, we are targeting the hardware generation for ultra low-power application domains; however for the sake of completeness, HLS tools targeting other design constraints are also briefly discussed in this action.

3.3.1 Multi-mode HLS

Kumar et al. [80] and Chavet et al. [21] targeted a different aspect of HLS applications that focuses on the reconfigurable multi-mode architectures. The multi-mode or multi-configuration architectures are specifically designed for a set of time-wise mutually exclusive applications (multi-standard applications).

Kumar et al. used FDLS algorithm for solving the scheduling problem while HAL is used for resource binding. On the other hand, Chavet et al. used list scheduling algorithm to perform scheduling while a maximal bipartite weighted matching algorithm for resource binding. Although, both of these approaches showed an impressive reduction in area-consumption, they are targeted toward the data-intensive DSP-systems.

3.3.2 Word-length aware HLS

Kum et al. [77] tackled another problem existing in embedded systems and combined the word-length optimization of DSP-operators with HLS. Their approach uses an MILP formulation for list scheduling algorithm to solve the scheduling problem while a heuristic-based maximum-*clique* partitioning algorithm to solve the register-binding problem. It also exploits the datapath merging technique (the concept of datapath merging was introduced by Van der Werf et al. [140]) and selects the largest word-length operator so that the smaller operations can also be mapped to the same operator if possible.

Figure 3.4 shows the basic concept of datapath merging where two different datapaths are merged to generate a single datapath circuit. The cost of this merging is

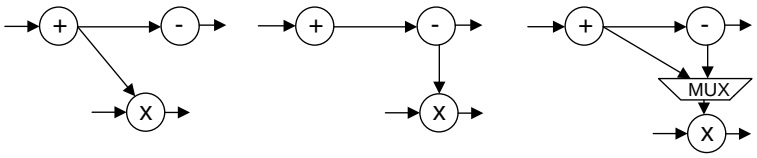


Figure 3.4: Dataflow graph (DFG) merging.

additional multiplexer as shown in the figure whereas the benefit is the reduction in the number of operators present in a reconfigurable system.

The solution by Kum et al. [77] is targeted toward a high throughput design rather than a low-power design. Nevertheless, it gave us an insight that the selection of an optimum word-length for operators could be helpful in reducing the area and power consumed by the hardware micro-task datapath. We have included a bitwidth adaptation stage in our design-flow that is discussed in Section 4.2.3.

3.3.3 Datapath-specification-based HLS

Unlike the above-mentioned HLS tools that are based on an iterative-improvement-based techniques, there is a class of HLS tools that take a datapath specification as input and generate the RTL-description in a single step. Two of such tools are discussed in the following sections.

3.3.3.1 User Guided HLS (UGH)

Augé et al. [10] proposed a design-flow for User Guided HLS (UGH) for the generation of coprocessors under timing and resource constraints. As compared to other HLS flows, UGH uses a single-step HLS where the datapath is pre-specified by the user at the input of the tool. Hence, the tool is suitable for VLSI designers who have a close knowledge of the actual datapath hardware. It uses two-step scheduling scheme that is further explained below:

- ***Coarse-grain scheduling (CGS)***: This step implements a list-scheduling-based solution to solve the issue of scheduling, resource allocation and binding.
- ***Fine-grain scheduling (FGS)***: FGS is performed after an early datapath and its control FSM has been generated. It basically performs retiming in the FSM to meet the actual physical delay constraints of the datapath components to achieve a desired frequency.

The UGH synthesis flow is presented in Figure 3.5. It is split into three steps. (i) The CGS generates a datapath and an FSM, called CG-FSM, from the C program and the datapath description. (ii) Then the mapping is performed. Firstly, the generation of the physical datapath is assigned to classical back-end tools using a target cell library. Secondly, the temporal characteristics of the physical datapath are extracted. (iii)

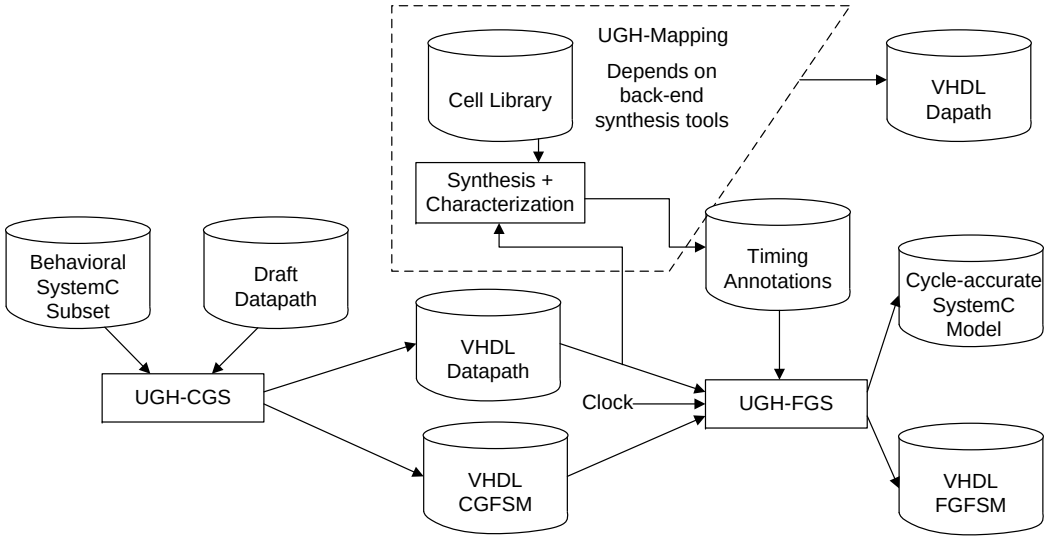


Figure 3.5: Complete UGH design flow [10].

Finally, the FGS retimes, for the given frequency, the finite control step machine taking accurately into account the annotated timing delays of the datapath and produces the FG-FSM of the circuit.

The notion of FGS and retiming of control FSM could be used in our system to address a different problem that is the optimization of hardware micro-task FSM by merging different FSM-states present in the basic block, of a CDFG, to have a possible reduction in number of FSM-states to reduce its power consumption.

3.3.3.2 No Instruction-Set Computer (NISC)

Reshadi et al. [42] proposed an approach that compiles a C program directly to a datapath and its controller. Since there is no instruction abstraction in this architecture they named it No Instruction-Set Computer (NISC). A datapath specification is provided at the input of the design-flow along with the application. The design-flow performs both scheduling and resource binding simultaneously through ALAP-like algorithm. The control-words generated to manage the datapath are kept in a control-memory. Similar to other HLS tools, the target applications are data-intensive algorithms. The area reduction is achieved by reducing the instruction fetch and decode stage of the processor while a large-size control memory can cause a significant amount of dynamic as well as static power consumption. Figure 3.6 shows the NISC design-flow.

3.3.4 Commercial tools and their application domain

After discussing the HLS tools present in the academics, we now present some of the industrial HLS tools and their potential applications. Recent generation of industrial

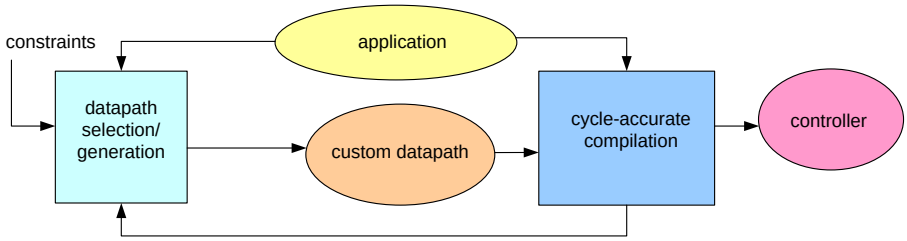


Figure 3.6: NISC design-flow [42].

HLS tools, in most cases, uses either ANSI-C, C++, or languages such as SystemC that are based on C or C++ that add hardware-specific constructs such as timing, hardware hierarchy, interface ports, explicit specification of parallelism, and others. Some HLS tools that support C or C++ or derivatives are Mentor's Catapult-C (C, C++) [92], Forte's Cynthesizer (SystemC) [40], NEC's CyberWorkbench (C with hardware extensions) [98], Synfora's PICO (C) [126], and Cadence's C-to-Silicon (SystemC) [16].

As far as the application domain of these HLS tools is concerned, examining the technical publications from Mentor Graphics on Catapult-C Synthesis, we found Nokia using HLS to generate hardware implementations of DSP algorithms for wireless communications. Alcatel Space also applied Catapult-C to DSP blocks for power, frequency, and timing recovery. From Forte's website, we see acknowledgments that Toshiba used the Cynthesizer for H.264 multimedia design. Summarizing these findings, we can see a lot of interest in using HLS for DSP blocks for wireless and wired communications and for image processing [47].

It can be clearly seen for both the academic as well as the commercial tools enumerated above that most of them (except for UGH) focus on data-dominated application domains where the hardware specialization is seen as a mean to improve system efficiency through parallelization. Thus, they are not suitable for control-oriented domains (such as ultra low-power WSN applications) where the focus is power-reduction, instead of an increased system throughput. In this thesis, we tried to fill the gap by introducing a design-flow that performs the hardware specialization for control-oriented applications.

In the following part of this chapter, we provide the existing design methodologies, algorithms and tools (both academic and commercial) for ASIP design.

3.4 Application Specific Instruction-set Processor (ASIP) design

An ASIP is a processor designed for a particular application or for a set of applications. It exploits special characteristics of application(s) to meet the desired performance, cost and power requirements. ASIPs fill the architectural spectrum between general-purpose

programmable processors and dedicated hardware or ASICs. They provide a compromise between the two approaches, i.e. high flexibility through software programmability (provided by general purpose MCUs) and high performance (high throughput and low energy consumption provided by ASICs).

To program an ASIP, the desired algorithm is written in a high-level language (C/C++). It is then translated by a compiler to generate a machine code that can be interpreted by the ASIP. The ASIP compiler needs the precise information about the underlying architecture of the processor and the algorithm must be written in such a fashion to facilitate the work of compiler. The ASIP synthesis tool must be capable of providing a fine blend among the algorithm, the compiler and the architecture to generate a successful ASIP.

Moreover, the ASIP design does not mean only to design the underlying architecture, but the ASIP designer must also come up with the corresponding software toolkit (e.g. the compiler, simulator, debugger etc.) that can help a user to benefit from the ASIP. To manually design these software toolkits is a time-consuming, expensive and error-prone process, hence, in the literature the researchers have always provided complete design methodologies to design their ASIPs. These approaches are based on Architecture Description Languages (ADL) that are used to describe the generic architecture of an ASIP and then the automatic design-flow goes down till hardware generation of ASIP along with its associated software toolkit. Some ADLs present in the literature are LISA [58], nML [36], ISDL [50], EXPRESSION [51] and Armor [93]. Most of these tools provide a path to retargetable ASIP compiler generation and only few of them target actual ASIP hardware synthesis.

Although, some ASIPs are still designed completely from scratch to meet extreme efficiency demands using the above-mentioned approach. There has also been a strong trend toward the use of partially predefined, configurable RISC-like embedded processor cores that can be quickly tuned to given applications by means of Instruction-Set Extension (ISE) techniques. This approach is helpful in reducing the design efforts and "time-to-market" while sacrificing some performance efficiency.

In the next part of this section, we briefly discuss the methodologies present in the literature for both the complete as well as partial ASIP design (using ISE).

3.4.1 Methodology for complete ASIP design

The ASIP design-flow normally follows a generic design methodology that has been presented in the survey paper by Jain et al. [68]. The authors have identified five major steps during the ASIP design that are briefly discussed in the following paragraphs.

- ***Application analysis***: Input to the ASIP design process is an application or a set of applications, along with their data and design constraints. The application is analyzed to gather the desired characteristics that can guide the hardware synthesis as well as instruction set generation. An application written in a high-level language (HLL) is analyzed statically and dynamically and the analyzed information is stored in some suitable intermediate format, which is used in the subsequent steps.

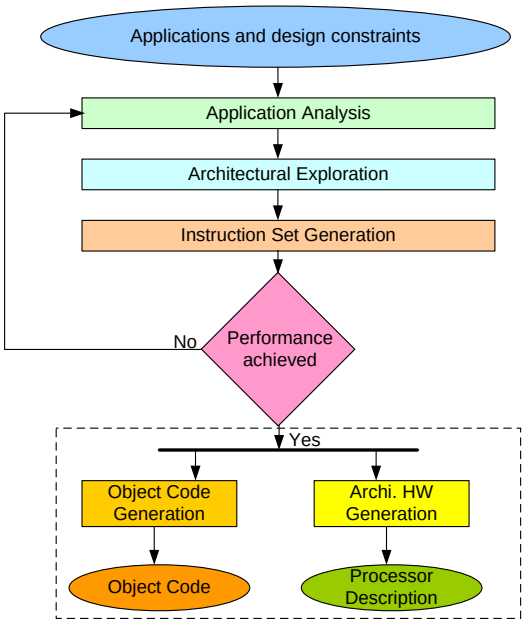


Figure 3.7: Design methodology for complete ASIP generation

- ***Architectural design space exploration***: First of all, a set of possible architectures is identified for a specific application(s) using the output of previous step as well as the given design constraints. Performance of possible architectures is estimated and suitable architecture satisfying performance and power constraints and having minimum hardware cost is selected.

- ***Instruction-set generation***: In the next step, instruction-set is generated for that particular application and for the architecture selected. This instruction set is used during the object code synthesis and hardware synthesis steps.

- ***Object code synthesis***: Compiler generator or retargetable code generator is used to synthesize code for the particular application(s).

- ***Hardware synthesis***: In this step the hardware is synthesized using the ASIP architectural template and Instruction-Set Architecture (ISA) starting from a description in VHDL/Verilog using standard tools.

Figure 3.7 shows the steps involved during the complete ASIP design. A performance evaluation phase exists between the instruction set generation step and code and hardware synthesis steps. This phase tests if the generated ASIP satisfies the desired design constraints or not. If yes, then we continue with code and hardware generation. Else, the process is iterated from the start.

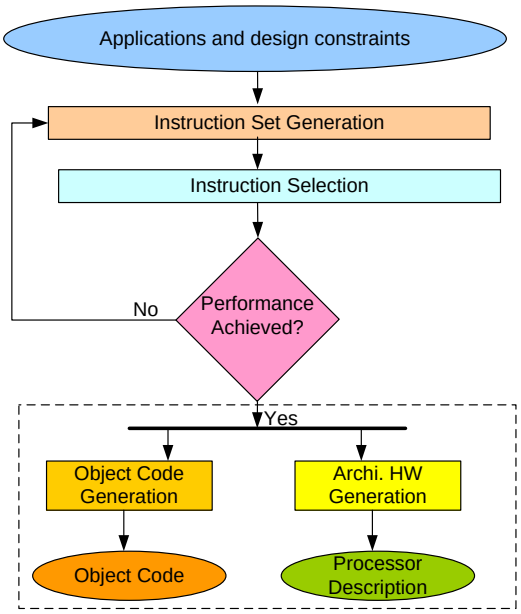


Figure 3.8: Design methodology for partial ASIP generation

3.4.2 Methodology for partial ASIP design

The complete ASIP design methodology consists in designing a complete processor along with its software toolkit to guarantee the harmony between the processor architecture and its compiler. However, the design-flow is not completely automated and certain critical parts of the processor have to be manually implemented as in case of design methodology using LISA [58]. Similarly, every new processor has to be validated and optimized that can cause Non-Recurring Engineering (NRE) costs. To reduce these costs, ISE technique is used where the instruction set of a predefined RISC-like processor is extended with special instructions that can be run on a coprocessor generated through the partial ASIP design-flow.

Figure 3.8 presents the main steps involved in the design-flow of an extended processor core generation. These steps are briefly discussed in the following paragraphs.

- ***Instruction generation***: In first step, the application code is analyzed and specialized instructions are generated for that particular application and for the architecture selected.
- ***Instruction selection***: In the next step, a sub-set of these special instructions is selected to efficiently implement the required function under the design constraints provided at the beginning. Like the design-flow for complete ASIP design, this process can be iterative until the required constraints are met.
- ***Object code synthesis***: Compiler generator or retargetable code generator is

used to synthesize code for the particular application(s).

- ***Hardware synthesis***: In this step the hardware is synthesized using the coprocessor architectural template and ISE starting from a description in VHDL/Verilog using standard tools.

The next section briefly presents the existing algorithms involved in different steps of ASIP design, such as instruction selection and register allocation.

3.4.3 Instruction selection

Instruction selection is the stage of a compiler back-end that transforms the Intermediate Representation (IR) (like CDFG), of an input application code, into a machine-specific IR that is very close to its final target language. In a typical compiler, it precedes register allocation, so its output IR has an infinite number of pseudoregisters; otherwise, it closely resembles the target assembly language. It works by *covering* the intermediate representation with *patterns*. The best *covering* is the one that results in the fewest *patterns* being generated for a given IR. A *pattern* is a template that matches a portion of the IR and can be implemented with a single machine instruction.

There are two approaches to perform the covering of an IR: (i) through instruction selection on trees and (ii) through instruction selection on Data Acyclic Graphs (DAGs). Instruction selection on DAGs is a much more computationally difficult problem than instruction selection on expression trees. In has been shown in literature that the former is an NP-Complete problem whereas the latter can be solved in polynomial time [76].

In the following section, we will first discuss the DAG-based instruction selection and existing algorithms to solve this problem. Tree-based instruction selection and its corresponding algorithms are discussed afterward.

3.4.3.1 DAG-based instruction selection

DAG-based instruction selection results in a much more efficient covering as the IR of an application code can be covered with a less number of patterns. For example, Figure 3.9 shows the basic *butterfly* operation performed in FFT (Fast Fourier Transform) algorithm. This operation has multiple outputs and thus it cannot be covered with a single tree-based pattern, whereas it is possible to cover it with a single DAG-based pattern. Hence, a DAG-based instruction selection results in a better covering with lesser number of output patterns. However, it is computationally much more complex to perform DAG-based instruction selection and lies in the domain of NP-Complete problems. Hence, several heuristic algorithms have been developed to provide an approximate solution.

In the following paragraphs, we discuss some of the existing algorithms that are being used for instruction selection on DAGs.

Simulated annealing: Simulated annealing is a meta-heuristic approach inspired by a process used in metallurgy. This process consists of alternating cycles of slow

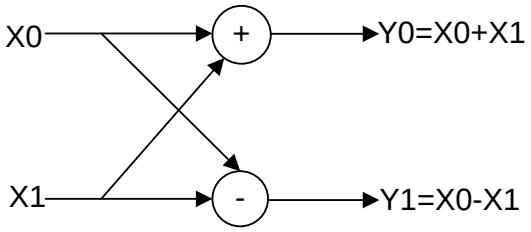


Figure 3.9: Data-flow graph of basic *butterfly* operation present in an FFT algorithm.

cooling and reheating to minimize the energy of the materials. Simulated annealing is, therefore, based on a parameter that represents the *temperature*. Similar to the physical process of annealing, the simulated annealing algorithm modifies one simulated solution to get another new one. If the new solution is better than the initial one, it leads to the local optimum. On the other hand, a worse solution can be used to explore more widely the solution space and saves us from being trapped in a local optimum. It can be thus interesting to keep a worse solution depending on a probability calculation that is based on the cost difference and the temperature of the system. Simulated annealing has been used by several previous works to perform the instruction selection [71, 60].

Genetic Algorithm (GA): GA belong to the family of evolutionary algorithms (a sub-set of meta-heuristics). Genetic algorithms are stochastic algorithms based on adaptive search for solving optimization problems. These algorithms are based on the theory of evolution and Darwin's principle of natural selection and apply them to a population of solutions possible to the problem. A genetic algorithm starts with a base population, generated randomly, among all solutions. Then this population undergoes cyclical stages of evaluation, selection, crossover and mutation. The algorithm stops after a given number of iterations.

Constraint Satisfaction Problem (CSP): Martin et al. [91] used CSP-solving based approach to resolve the problem of instruction selection for ASIP ISE generation. A CSP is defined as a 3-tuple $\mathtt{S=(V,D,C)}$ where $\mathtt{V=x_1,x_2,x_3,\ldots,x_n}$ is a set of variables, $\mathtt{D=D_1,D_2,D_3,\ldots,D_n}$ is a set of finite domains (FDs), and $\mathtt{C}$ is a set of constraints.

A solution to a CSP is an assignment of a value from variable's domain to every variable, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need just one solution, all solutions or an optimal solution given some cost function defined in terms of the variables.

The solver builds, using the given constraints, its own consistency methods and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Solutions to a CSP are usually found by systematically assigning values from variables domains to the variables. It is implemented as depth-first search.

The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

3.4.3.2 Tree-based instruction selection

Though the above-mentioned approaches that perform instruction selection on DAGs such as [76, 91] result in a better covering of the CDFG (i.e. lesser number of resultant patterns), the classical approach to instruction selection has been to perform tiling (pattern matching) on expression trees. The problem is simpler in complexity and can be solved in polynomial time. The problem was initially resolved using dynamic programming that is discussed in the following paragraph.

Dynamic programming: Dynamic programming resolves a problem by combining solutions of sub-problems. Dynamic programming is interesting especially when the sub-problems are not independent, i.e. when sub-problems have common sub-sub-problems. A dynamic programming algorithm resolves each sub-sub-problem once, and memorizes the solution. A recalculation of the solution does not take place whenever the sub-problem is encountered. Dynamic programming is thus an interesting approach when sub-problems have many sub-sub-problems in common. Most of the existing solutions to the instruction selection on expression trees are based on dynamic programming [3, 41, 114].

The instruction matching techniques on expression trees have been further developed to yield code-generator generators [41] that take a declarative specification of a processor architecture at the input and, at compiler-compile time, generate an instruction selector. These code-generator generators either perform dynamic programming at compile time [3] or use BURS (Bottom-Up Rewrite System) tree-parsing [114] to move up the dynamic programming to compiler-compile time.

Bottom-Up Rewrite System (BURS) generator: Possibly the simplest way to visualize and understand complex instructions and addressing modes of a processor is to view them as expression trees in which leaves represent registers, memory locations, or constant values; whereas internal nodes represent operations. Describing even the most complex instruction is simplified when such trees are used. Figure 3.10 gives an example of tree patterns whereas Figure 3.11 shows the two possible coverings of an identical tree with different patterns.

Code generators based on BURS can be extremely fast because all dynamic programming is done when the BURS automaton is built. At compile-time, it is only necessary to make two traversals of the target expression-tree: (i) bottom-up traversal to label each node with a state that encodes all optimal matches and (ii) top-down traversal that uses these states to select the instructions for each node according to the minimal cost function and emits target assembly level IR.

Pattern #	Label → Pattern	Cost
1	goal → reg	(0)
2	reg → Reg	(0)
3	reg → Int	(1)
4	reg → Fetch(addr)	(2)
5	reg → Plus(reg, reg)	(2)
6	addr → reg	(0)
7	addr → Int	(0)
8	addr → Plus(reg, Int)	(0)

Figure 3.10: Sample machine instruction template.

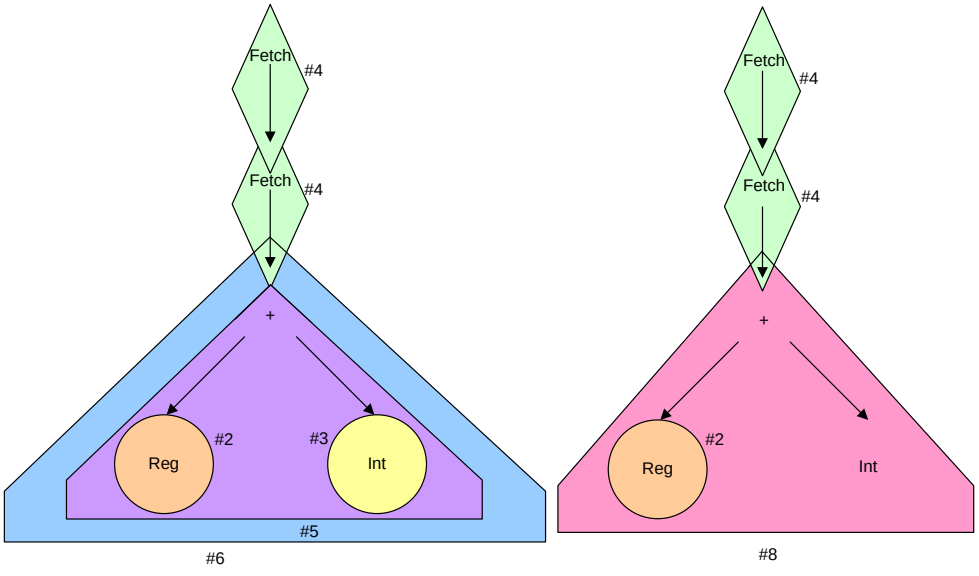


Figure 3.11: Two possible coverings of the identical tree with different patterns.

Rule #	Simple Grammar L.H.S → R.H.S.	Cost
1	goal → reg	(0)
2	reg → Reg	(0)
3	reg → Int	(1)
4	reg → Fetch(Addr)	(2)
5	reg → Plus(reg,reg)	(2)
6	addr → reg	(0)
7	addr → Int	(0)
8	addr → Plus(reg, Int)	(0)

Figure 3.12: Simp1e grammar and its normal form [114]

The input to the BURG-generator is a set of rules. Each rule is a quadruplet of the form $R = (P, S, C, A)$ where P is a pattern existing in the IR, S is the replacement symbol, C and A are the cost and action taken if the given rule is selected.

To further elaborate, S is a *nonterminal* symbol whereas A can be, for instance, the resultant machine instructions generated if the concerning rule is selected for the given pattern. In addition the rules are defined by keeping in mind the underlying hardware of the datapath executing these instructions. Figure 3.12 gives a sample grammar. The nonterminal is on the left side of the rule, the linearized tree pattern is on the right side. In this grammar, `goal`, `reg`, and `addr` are nonterminals. In addition to nonterminals, the grammar has *operators*. For instance, `Reg`, `Int`, `Fetch`, and `Plus` are some operators present in the sample grammar.

A least-cost parse can be found using dynamic programming. By using all matching patterns at all nodes, it is possible to remember the rules that lead to the least-cost derivation for each possible nonterminal. Figure 3.13 applies the rules in Figure 3.12 to a tree representing `Fetch(Fetch(Plus(Reg, Int)))`. We have only highlighted the possible matches present at `Reg` and `Int` nodes.

In this example, a BURS matcher finds the least-cost parse of an expression tree to reduce all the nodes to the `goal` nonterminal. Each node is labeled with a state that contains all the rules that lead to the reduction of this node to all the possible nonterminals. For example, it is possible to reduce the node `Int`, to all the nonterminals. `Int` can be reduced to the nonterminals `reg` and `addr`, by directly applying the rules 3 and 7 respectively. The costs associated with each derivation is the cost of that particular rule. The reduction toward `goal` utilizes the rule, "`goal` → `reg`" that will require that `Int` to be subsequently reduced to `reg`. Therefore, while the cost associated with rule 1 is 0, the total cost of the reduction from `Int` to `goal` is 1.

3.4.4 Register allocation

The resource allocation problem relates to simple register allocation in ASIPs that have single processing element, while in ASIPs that have Very Large Instruction Word

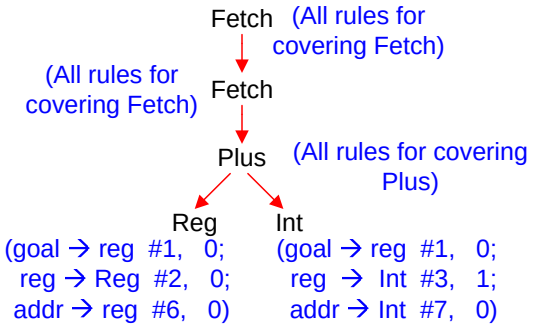


Figure 3.13: Dynamic programming applied to example tree, each node labeled with "(Rule, Cost)".

(VLIW) architecture, it also consists in functional unit allocation. As we described in previous section that instruction selection results in an assembly-level IR that contains infinite number of pseudoregisters. The allocation of the limited number of physical registers present in the register file of an ASIP to these infinite pseudo-registers is called *register allocation*. Similar algorithms are used in ASIP design-flow for register allocation as those were used in HLS design-flow (see Section 3.1.4).

After a thorough description of ASIP design-flows and their related algorithms, we move on to present some existing work in the domain of ASIP design tools.

3.5 Existing tools in ASIP design

There exist in the literature several commercial as well as academic examples of configurable ASIPs. Xtensa by Gonzalez is a configurable and extensible coprocessor core that builds around a traditional RISC five-stage pipeline [46, 128]. Silicon Hive is a commercial tool that generates a framework of multiple ULIW (Ultra Large Instruction Word) coprocessors [124]. Virage Logic [141] has also launched a series of commercial ASIPs like ARC750, ARC700 etc. In addition, different FPGA vendors have launched their specialized soft-cores that can be implemented on their FPGAs. Most famous of these FPGA-based soft-cores are NIOSII by Altera [6] and MicroBlaze by Xilinx [144]. Aeroflex Gaisler [2] has introduced LEON series of soft-cores that are based on SPARC V8 processor architecture. The FPGA-based soft-cores provide flexibility and can be used for rapid prototyping. However, evidently they are consume more power and provide less performance as compared to ASIC-based ASIPs.

3.5.1 ICORE

As far as low-power ASIP development is concerned, Glöker et al. proposed ICORE [45], a low-power ASIP specialized for the data-intensive application of a DVB-T receiver. Their design methodology starts with a semi-custom design and instruction-set en-

hancements (hardware acceleration) are applied to this unoptimized architecture to achieve a desired speed-up. When all the timing constraints are fulfilled, additional optimizations are applied to increase the power-efficiency. The authors used two simple rules to apply these optimizations: (i) they tried to find a power optimized (hardware accelerated) solution for the most frequently used patterns, (ii) if there existed a simple hardware solution for even a less-frequent task, it was still implemented in hardware to save power. They synthesized an ICORE architecture using 0.18 μm 5-metal layer technology. ICORE was able to get a speed-up of nearly 15x when compared to traditional DSP with only a slight increase in average power whereas the overall energy is reduced by 92.5% as the total time taken to do the job is considerably reduced.

On the other hand, Kin et al. [74] also proposed a framework for rapidly exploring the design space of power-efficient mediaprocessors in the domain of VLIW and SIMD (Single Instruction Multiple Data) architectures. Their framework is based on the idea of using a retargetable Instruction Level Parallelism (ILP) compiler and its corresponding processor simulator to perform rapid prototyping of media application benchmarks. They used StrongARM SA-110 as their baseline architecture and their framework exploited the ILP compiler to perform the design space exploration of multi-processor platforms to generate power-efficient mediaprocessors.

3.5.2 Soft-core generator

Fin et al. [37] introduced the concept of soft-core generation by instruction-set analysis. The soft-core generator can be easily applied for parameterization and designing any kind of processor which can be described by using FSMDs (FSMs with datapaths). By using different parameters at the input of the soft-core generator design-flow, several soft-cores performing the selected instruction sub-sets are generated. The VHDL codes for these soft-cores are then easily synthesized to find the area-reduction caused by the specialization process.

3.6 General discussion

Interestingly, all these ASIP design and HLS tools share a common characteristic: they generate the hardware that is specialized toward data-intensive applications and they generally see hardware specialization as a mean to improve performance over a standard software implementation. This performance improvement, however, often comes at a price of increased area cost (coprocessor or ISE requires additional area). Of course, these specializations also have a significant impact on power efficiency, since they allow for a drastic reduction of dynamic power of the system, however, the overall static power dissipation of the system is increased due to the increase in overall silicon area. This increased static power can be crucial for a power-restricted application domain such as WSN.

Indeed, except from [37] and [86], very few papers have addressed the problem of using processor specialization as a mean to reduce silicon footprint. We believe that in the context of WSN node architecture, where silicon area and ultra low-power are the

two main design issues, such an approach deserves attention. In the next chapter, we present an original approach which builds on this idea. We propose a new architectural model (along with a complete system-level downto RTL design-flow) for WSN node controller design, based on the notion of *concurrent* power-gated hardware micro-tasks.

Chapter 4

Hardware micro-task synthesis

This chapter discusses in details our proposed design-flow for hardware micro-task synthesis as highlighted in Figure 4.1. The chapter starts with the notion of hardware micro-task where we provide the potential power benefits and architectural details of the proposed hardware micro-task. Our proposed design-flow for C to RT-level VHDL description of hardware micro-task synthesis is discussed afterward. Incidentally, we would like to clear it out that we do not propose new algorithms to solve different HLS/ASIP-design related problems such as instruction selection or register allocation, instead we use classical algorithms to solve these problems. The choice of using classical tools is mainly driven by our application domain that consists in control-oriented tasks of a WSN node. Since the target applications are not data-intensive, working on more sophisticated algorithm development would not be more beneficial. The chapter concludes with an example of a control-oriented micro-task (processed through our tool) showing the benefits of our approach over a conventional MCU-based software implementation.

4.1 Notion of hardware micro-task

In a typical WSN node, each task present in the Task Flow Graph (TFG) is handled by an MCU and corresponding OS that provides support for multi-tasking features. We call such task, a *micro-task*, to highlight the fact that it has light-weight processing requirements. In addition, these micro-tasks normally have a *run-to-completion* semantic. Figure 4.3 shows the generic template of a micro-task. In our approach, each micro-task present in the TFG is executed by a specialized hardware entity. This entity is called a "hardware micro-task".

4.1.1 Potential power benefits

Some of the important aspects of the hardware micro-task design that impact its overall power and energy consumption are discussed in the following section.

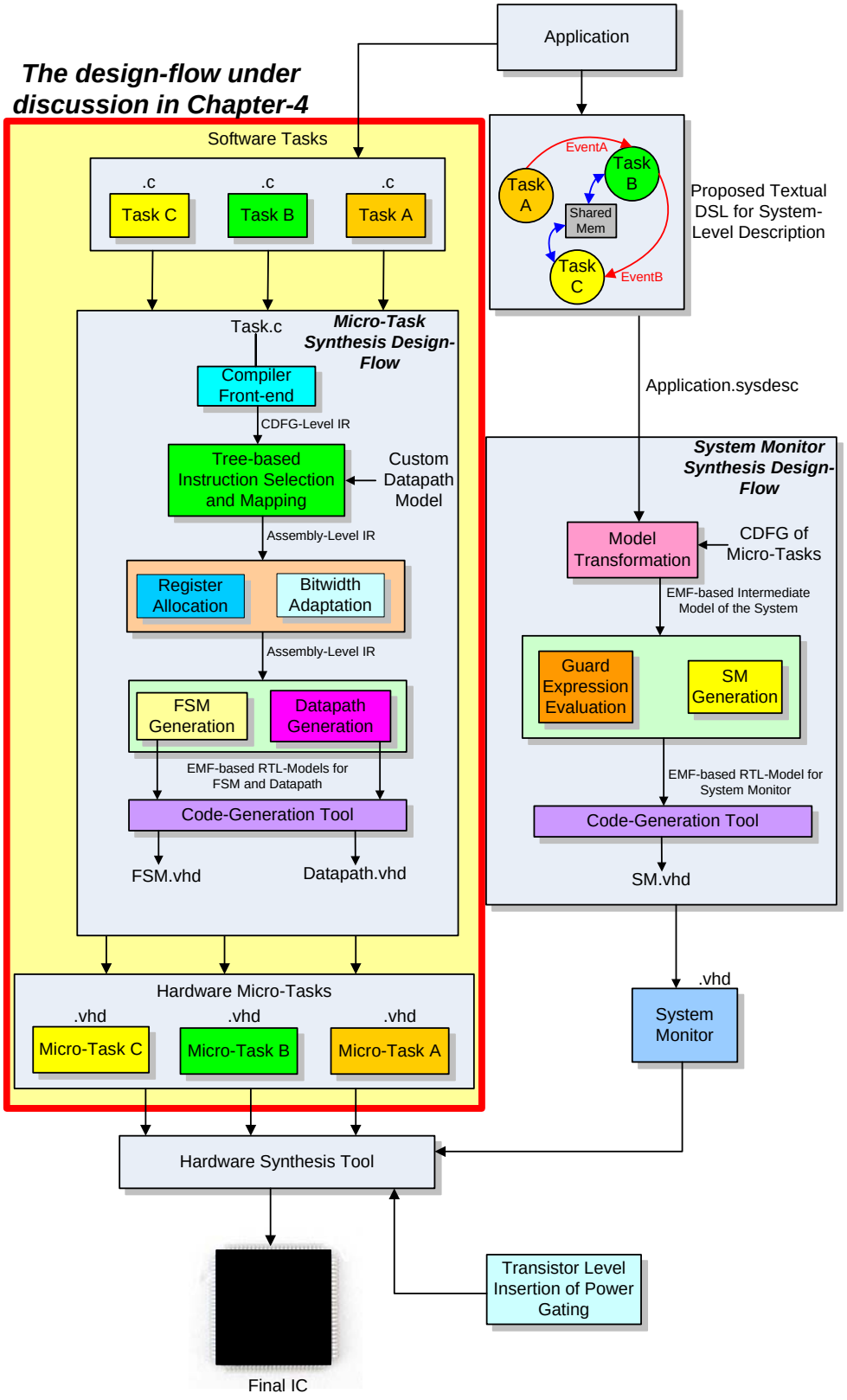
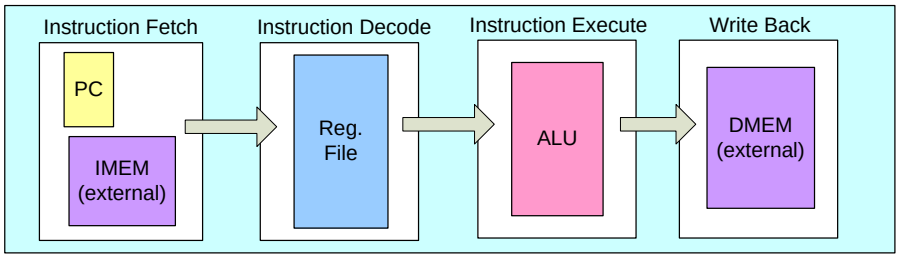
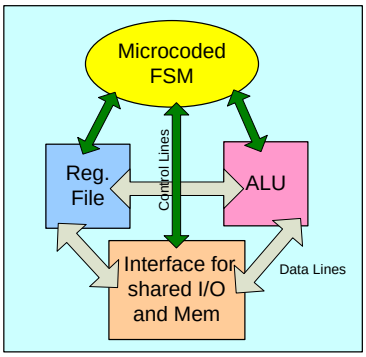


Figure 4.1: Design-flow for hardware micro-task generation.



(a) A simplified version of a general purpose CPU architecture with external instruction and data memories (*all the components are of fixed size and cannot be customized)*



(b) A simplified version of a generic hardware micro-task architecture (*all the components can be customized to save power and area)*

Figure 4.2: Architectural simplicity of a hardware micro-task w.r.t. a general purpose CPU.

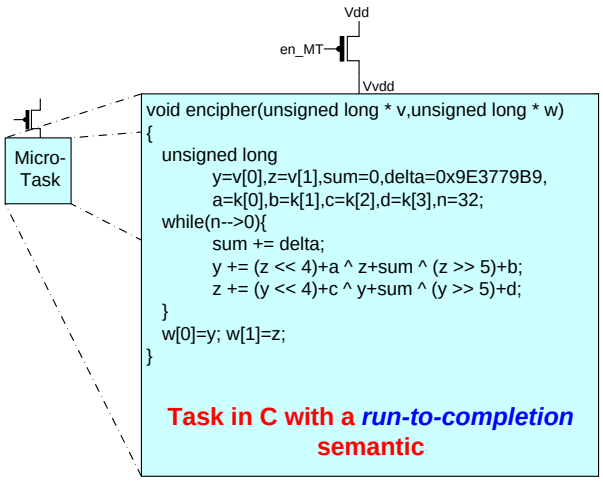


Figure 4.3: Generic template of a "micro-task" running in a WSN node.

4.1.1.1 Simplified architecture

In contrast to an instruction-set processor, the program of a hardware micro-task is hard-wired into an FSM that directly controls a semi-custom datapath. This makes the architecture much more compact (no need of an instruction decoder or instruction memory) and allows the size of storage devices (register file and ROM) as well as the ALU functions to be customized to the target application. Each of these hardware micro-tasks can access shared data memory and I/O peripheral (e.g. SPI link to an RF transceiver). The simplicity of the hardware micro-task architecture w.r.t. a generic architecture of an MCU (currently being used in WSN node design) can be seen in Figure 4.2. These general purpose MCUs are not customized to the application at hand and of course contain an instruction memory, fetch and decode stage. Whereas, our datapath contains neither instruction memory nor fetch/decode stage, and it can be customized to the application at hand. This leads to a reduction in both dynamic and static power consumption.

Such a drastically simplified architecture allows for obvious dynamic power savings compared to a standard MCU architecture, with a negligible loss in performance since the datapath is tailored to the application at hand.

4.1.1.2 Exploiting the *run-to-completion* semantic

As mentioned earlier, the micro-tasks follow a *run-to-completion* semantic. This leads to a stateless execution of the micro-tasks in which we do not need to store the state of the local variables present in the C-specification of a micro-task after its execution. In hardware implementation, it means that the whole micro-architecture of the hardware micro-task can be power-gated after its job-termination that leads to a lower *static power* dissipation.

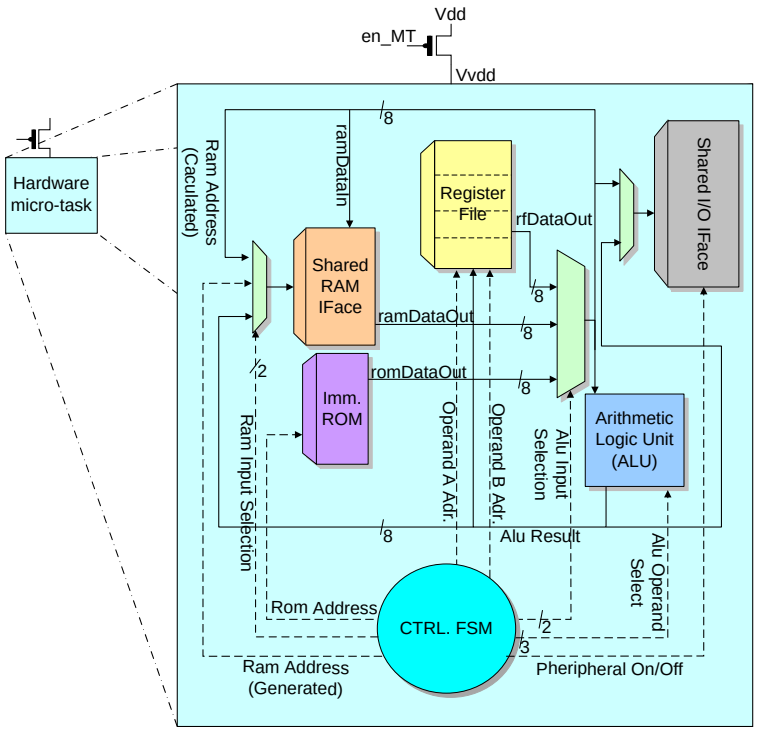


Figure 4.4: Architecture of a generic hardware micro-task.

4.1.1.3 Micro-task granularity

Hardware specialization can be less effective if more and more functionalities are handled by the same specialized hardware. As a consequence, to take advantage from hardware specialization, we need to distribute the whole WSN node software framework into a set of hardware micro-tasks, so as to maintain a high degree of specialization within each micro-task.

For instance, a complete WSN communication stack uses approximately 3500 instructions; by distributing the stack functionality onto 7 micro-tasks, we can reach an average micro-task code size of 500 instructions, which is a granularity level at which we can expect significant energy improvements. Since the micro-tasks present in WSN applications are fine-grain in nature with an average code size much smaller than 500 instructions (as shown in Chapter 6), we benefit from a higher degree of hardware specialization.

4.1.1.4 Simplified access to shared resources

It is also worth-mentioning that the local variables present in the C-specification of a micro-task are stored in the register file while the global variables are stored in external memory blocks that can be shared between multiple hardware micro-tasks. Depending upon the life-time of these global variables, these external memories can be power-gated

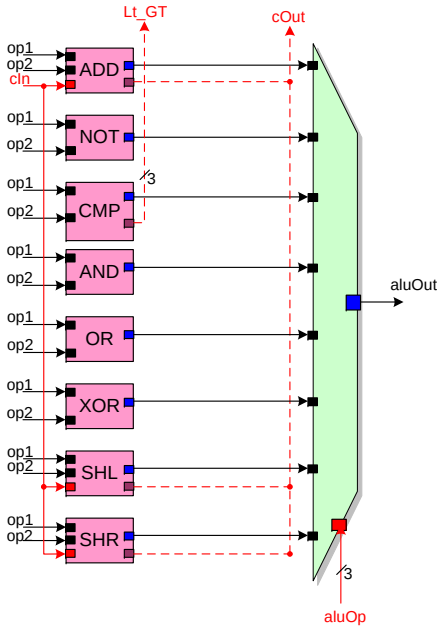


Figure 4.5: Architectural template of customizable ALU block present in hardware micro-task datapath (shown in Figure 4.6).

to save the leakage power. More details about the memory management are provided in Section 5.4.4 where the system-level execution-model is presented. Moreover, since the hardware micro-tasks are power-gated (turn-on only when needed), we do not need arbiters, tri-state or multiplexer logic at the input of the shared resources. This also leads to an overall reduction in power and area.

4.1.2 Generic architecture

Figure 4.4 shows the micro-architecture of a generic hardware micro-task. In this figure, the hardware micro-task consists of an 8-bit datapath whereas our design-flow is capable of generating both 8-bit and 16-bit datapath according to designer's choice. The possible trade-offs in terms of energy, power and area consumption of an 8-bit and a 16-bit hardware micro-tasks implementing the same function are discussed in Section 6.4.1.

The main components of a hardware micro-task are:

- ***FSM***: The control part of the application task is directly micro-coded in the form of an FSM that controls the underlying datapath.
- ***Register file***: The register file is implemented in the form of a dual-port RAM. The size of the register file can be customized according to the application by the design-flow. It contains two *read*-ports and one *write*-port that enables *reg-reg*-type instruction patterns to be executed in one clock-cycle by our datapath.

A hardware micro-task having a dual-port memory benefits from approximately 50% reduction in the size of the control FSM as compared to a hardware micro-task having a single-port memory. This potential reduction comes from the fact that a single-port memory has to be accessed twice consecutively while fetching two operands from the register file. Hence, if size of the control FSM is very large and its power consumption is dominant, a hardware micro-task containing a dual-port register file would consume lower power than a hardware micro-task containing a single-port register file. In addition, since the size of the register file needed by a hardware micro-task is relatively small (as it will be discussed in Section 4.2.4), it does not consume huge power in the resultant circuit.

- ***Immediate ROM***: The datapath also contains a ROM that stores all the constants present in the application code. Since the constants stored in ROM are hardwired in the hardware and the ROM can be turned-off without data loss, this approach consumes less static and dynamic power for storing constants.

- ***ALU***: The ALU block contains several arithmetic and logic operations (such as `add`, `sub`, `or`, `shl`, etc.). A generic template of the ALU block implemented in our hardware micro-task is shown in Figure 4.5 where different operators can be added/removed and size of the multiplexer is customized according to the application at hand.

- ***I/O interface***: The final major component of the hardware micro-task is an I/O interface module that provides the interface to external data memories and I/O peripherals (that can be possibly shared among multiple micro-tasks).

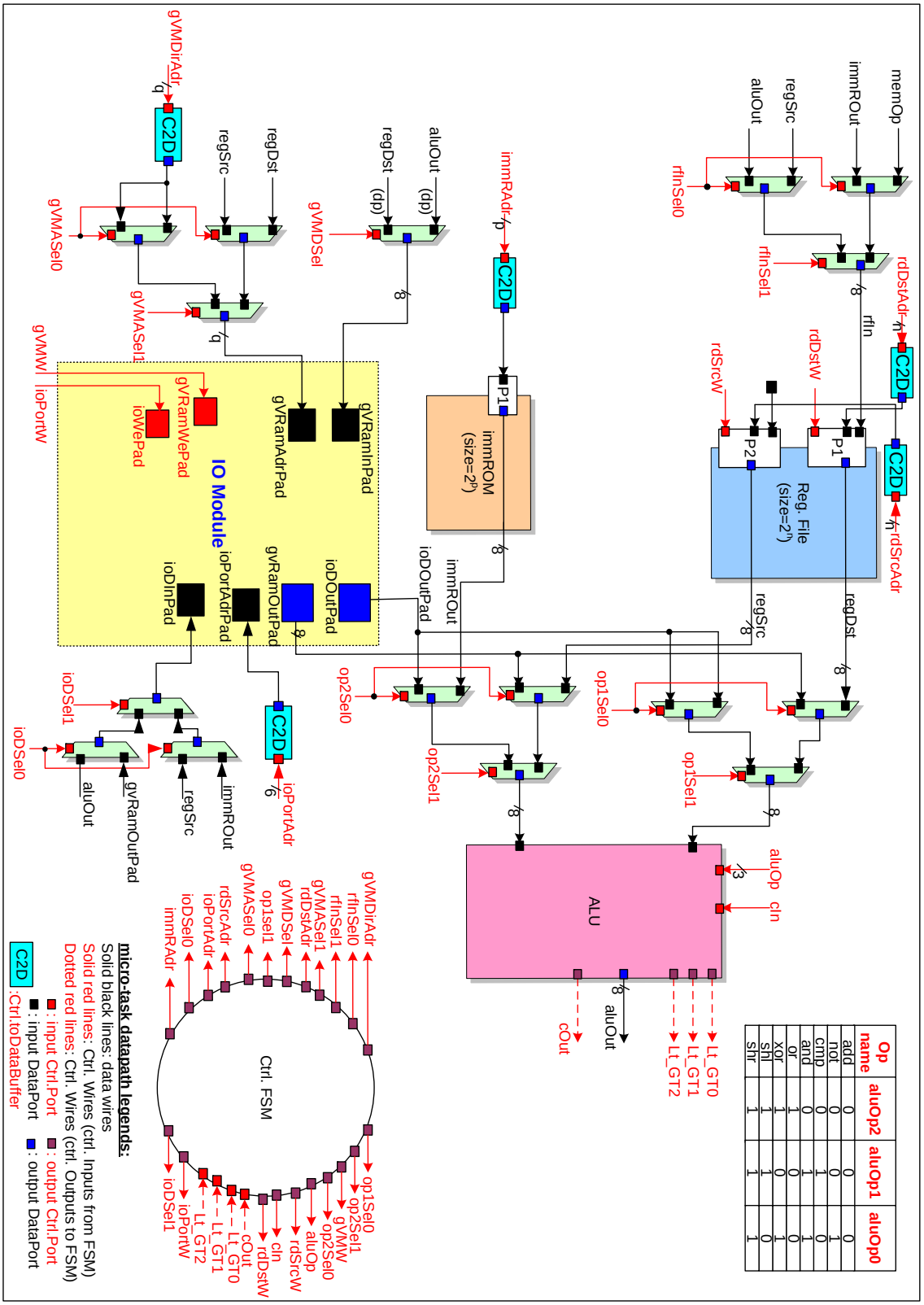
According to the legend of Figure 4.4, dotted lines represent control signals generated and exchanged between the control FSM and datapath components, whereas solid lines represent data-flow connections between datapath components.

A more detailed view of the hardware micro-task architecture is shown in Figure 4.6 where we can find different control-flow multiplexers (such as those connecting different types of operands to the ALU block).

To generate such architecture of a hardware micro-task from a high-level C-specification of the application, we developed a design-flow that is a hybrid of High-Level Synthesis (HLS) and retargetable Application Specific Instruction-Set Processor (ASIP) design-flows.

4.2 Proposed design-flow for micro-task generation

Of course, even the soundest proposal for hardware specialization is useless without a supporting design-flow, which allows the programmer to proceed directly from a specification written in a high-level language (e.g. C) to an executable specification, which in our case consists of an RTL description of the each specialized hardware micro-task.



Op name	aluOp2	aluOp1	aluOp0
add	0	0	0
not	0	0	1
cmp	0	1	0
and	0	1	1
or	1	0	0
xor	1	0	1
shl	1	1	0
shr	1	1	1

Figure 4.6: Detailed architectural template of a hardware micro-task.

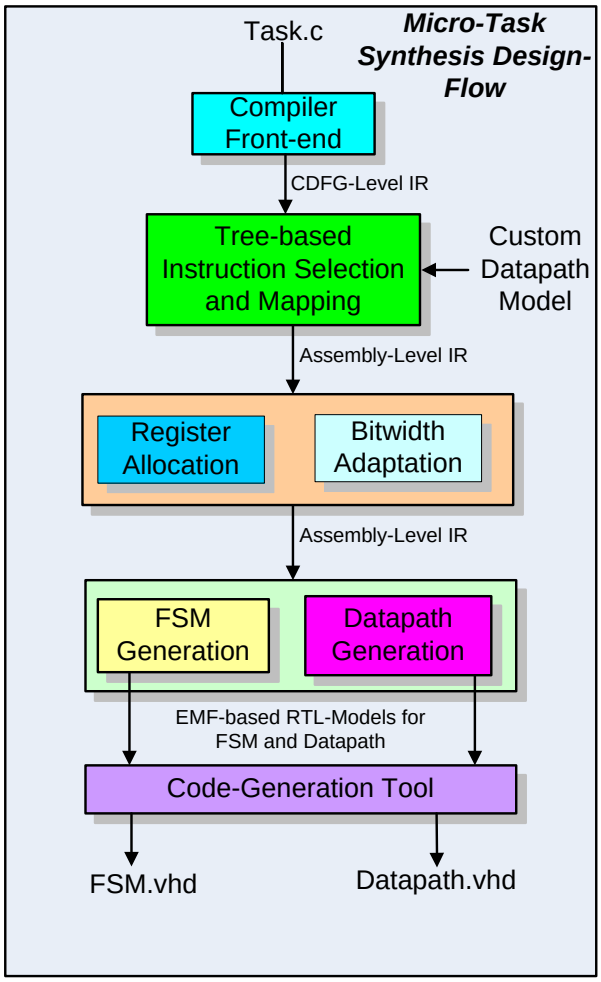


Figure 4.7: Design methodology for hardware micro-task generation.

This section details our proposed software design-flow used to generate these customized hardware micro-tasks from an application description written in C (shown in Figure 4.7). A comparison of our approach to classical HLS/ASIP-design flow is presented later in this section. We have identified six distinct steps involved in the design-flow for micro-task synthesis that are explained in the following sections.

4.2.1 Compiler front-end

Our flow begins with the front-end compilation of the ANSI-C specification of an application. This first step transforms the input description into a formal Intermediate Representation (IR). This step benefits from several target-independent code transformations such as constant evaluation and propagation, single static assignment, loop unrolling, etc. The output of this step is an IR (that is in the form of a CDFG) in

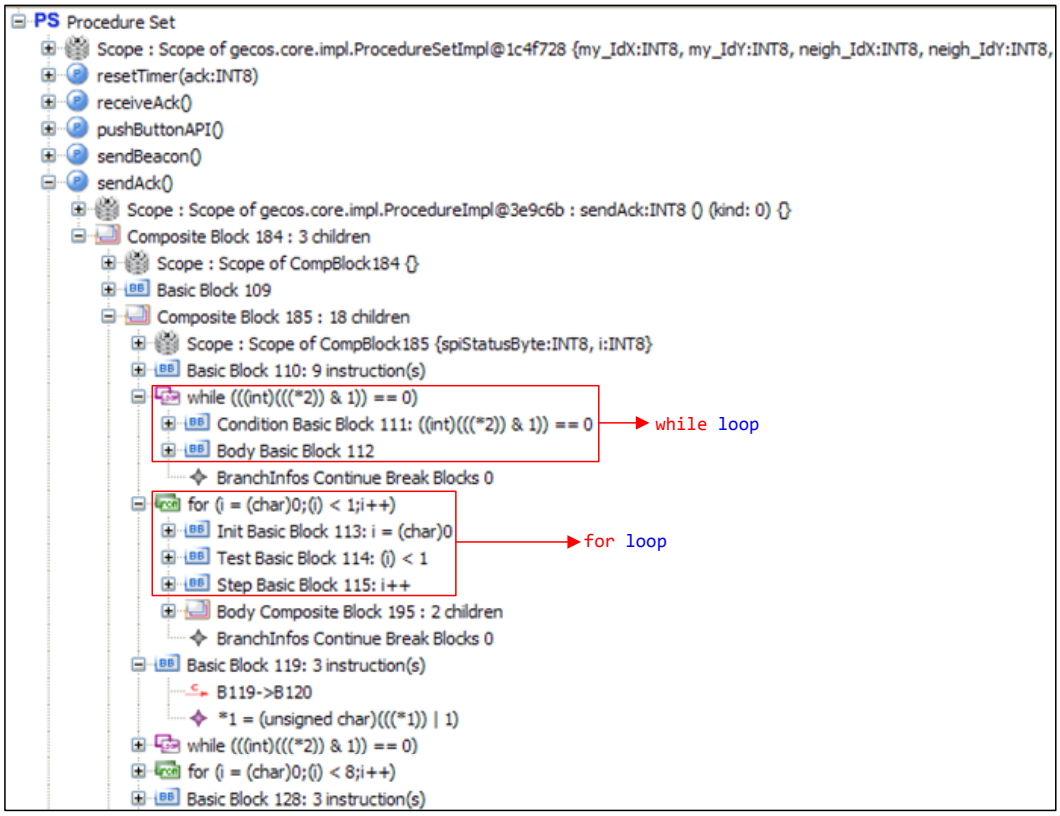


Figure 4.8: Example of a CDFG generated through GeCoS [86].

which instructions are represented as trees. This part of our design-flow is based on the front-end of the GeCoS [86] retargetable C-compiler, initially developed by L'hours. Figure 4.8 presents a generic example of the CDFG generated by GeCoS front-end where different sub-components like basic blocks, composite blocks, while, if, and for blocks can be observed. GeCoS can represent instructions present in a basic-block either as expression-trees or DAGs. In this work, we use tree-based intermediate representation.

4.2.2 Instruction selection and mapping

The tree-based IR represents each basic operation (e.g. memory fetch or store, addition or subtraction, conditional jumps, etc.) by a tree node. A real machine instruction often performs several of these basic operations at the same time. For example, a *mac* instruction can perform addition and multiplication in a single step. On the other hand the situation can be reversed i.e. a basic operation in an IR (e.g. a 32-bit memory fetch) can be mapped to a sequence of several machine instructions (4x 8-bit load instructions). Finding the appropriate mapping of machine instructions to a given IR-tree is done through an instruction selection phase.

As mentioned previously, even if there exist more sophisticated approaches for in-

```
/* The syntax of a BURG rule defined in our BURG generator is:
   Symbol : Pattern {Cost computation} = {Action} */
stmt: SET(REGISTER, ADD(reg8, mem))
{
        if (!isByteType($value[4]))
                return false;

        // this expression defines the cost (C) of this BURG rule
        $cost[0] = $cost[4] + $cost[5] + 2;

} = {   // this block defines the action (A) of this rule

        AsmInst add = new AsmInst ("addBG", 0, 3, false);
        add.addOperand ($action[4] ());
        add.addOperand ($action[5] ());
        block.addInstruction(add);
        return;
};
```

Figure 4.9: A sample BURG rule being used in our BURG-generator.

struction selection (e.g. instruction selection on DAGs [76, 83, 91]); our current implementation uses a simple BURG-based tree-covering algorithm to provide a polynomial time solution to the instruction selection problem as we target control-oriented applications where there is very little instruction-level parallelism. Moreover, the polynomial-time solution of instruction selection problem makes the tool useful for real-life applications that is also an important factor in our point of view. The approach is based on the work of L'hours who used the tree-based pattern matching to generate retargetable ASIP instruction-set architectures (ISAs) [86].

4.2.2.1 Customized BURG-generator

Following a similar approach as was used by Proebsting in [114], we used a back-end compilation tool called BURG-generator. The input to the BURG-generator is a set of rules of the form $R = (P, S, C, A)$ where P is a pattern existing in the CDFG (IR), S is the replacement symbol, C and A are the cost and action taken if the given rule is selected. Figure 4.9 shows a sample BURG rule that is being used by our BURG-generator.

Each of the terms used in a BURG rule and its difference from the conventional BURG-generator approach is explained in the following paragraphs.

***P*, *the pattern*:** Each rule in the BURG-grammar defines an instruction pattern that can be used to cover a given expression-tree if that rule is selected. Since most of the workload of a traditional WSN-based MCU consists of the communication with its RF-transceiver or sensor through SPI-interface [119], we define such a template for hardware micro-task datapath where memory and I/O modules can directly work as operands for ALU and direct interaction is possible between register-file, data memory

that carries the global shared variables, I/O modules and immediate ROM that carries the constant values (as shown in Figure 4.6). In order to better exploit this datapath, we do not limit ourselves to simple ISAs (such as MIPS, RISC or mini-MIPS) but generate relatively complex instruction patterns and their corresponding rules in our BURG-generator, so as to obtain an efficient covering.

***S*, the replacement symbol:** The second entity defined in a rule is S, the replacement symbol. S is normally a *nonterminal* and an instruction pattern described in a given rule is reduced to the *nonterminal* defined by that rule. We differ, in the definition of the replacement symbol, than the normal BURG-based instruction selection as we introduce *typed-nonterminals* that result in the generation of an assembly-level IR having *typed-pseudoregisters*. The *nonterminals* used in our grammar are `stmt`, `reg8`, `reg16`, `reg32`, `mem`, and `cond`. Apart from *nonterminals*, our grammar contains *terminals* that represent the *operator*-nodes in an expression tree. Some examples of these *terminals* are `SET`, `ADD`, `SUB`, `XOR`, `AND`, `OR` and `CMP` that correspond to assignment, addition, subtraction, exclusive-OR, AND, OR and comparison operations in a tree-based CDFG.

***C* and *A*, the cost and the action:** C and A are the cost and resultant action taken, if the given rule for instruction selection is used. A can be, for instance, the resultant machine instructions generated if the concerning rule is selected for the given pattern.

As the operating frequency for WSN applications is quite low (current low-power MCUs work at around 1 MHz to 4 MHz), the time available during a single clock-cycle is long enough to perform several operations. We exploited this fact and our instruction selection tries to minimize the total number of clock-cycles consumed during the execution of an input IR instruction.

Since the complex patterns mentioned-above involving the *memory*- or *I/O*-operands result in a lesser number of overall clock-cycles, the code selector generates these specialized instructions (involving *mem*- and *I/O*-operands), instead of generating multiple simple *reg-reg*-type instructions that involve additional *load* and *store* instructions to move the data from I/O-peripherals and memories to the register file.

The idea is illustrated with the help of an example in Figure 4.10. The input CDFG pattern is `SET(INDIR(INT),AND(INDIR(INT),INT))`. This pattern can be covered by multiple rules present in our BURG-grammar. One of these rules corresponds to a complex pattern, which can be executed by the hardware micro-task datapath in only two clock-cycles (as shown in Figure 4.10, Case A). The same input pattern could have been covered by simpler patterns that would have resulted in a sequence of *reg-reg*, *load* and *store* instructions consuming 5 clock-cycles (as shown in Figure 4.10, Case B).The example clearly shows that the *selected* specialized pattern has a lower cost in terms of execution time.

As mentioned earlier, the BURG-rules are defined according to the underlying

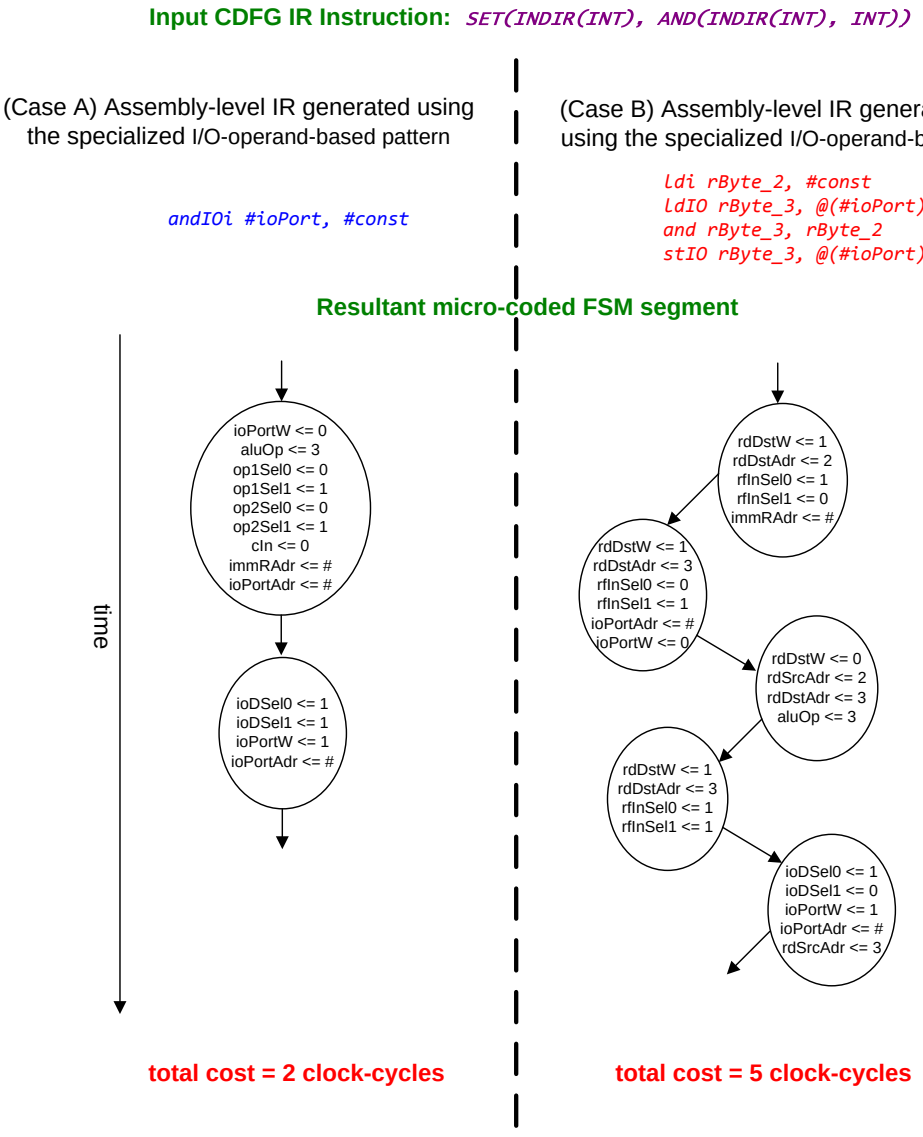


Figure 4.10: Advantage of using specialized pattern that results in an overall reduction in cycle-count.

S	P	C	A	Comments
stmt	*SET(INDIR(INT), AND(INDIR(INT), INT))*	2	`andIOi #ioPort, #const`	Performs an AND operation of an I/O port and a constant value
stmt	*SET(INDIR(INT), INT))*	1	`movIOi #ioPort, #const`	Moves a constant value to an I/O port
stmt	*SET(INDIR(INT), OR(INDIR(INT), reg8))*	2	`orIOB #ioPort, rByte_2`	Performs an OR operation of an I/O port and an 8-bit variable
stmt	*SET(REGISTER, XOR(reg8, INT))*	2	`xoriB rByte_3, #const`	Performs an EX-OR of an 8-bit variable and a constant value.
stmt	*SET(REGISTER, AND(reg8, mem))*	2	`andBG rByte_2, @(rByte_3)`	Performs an AND operation of an 8-bit variable and the contents of a memory location
mem	*INDIR(reg16)*	1	`@(rInt_5)`	Covers a memory access by a 16-bit pointer
reg16	*AND(reg16, reg8)*	2	`andIB rInt_3, rByte_5`	Performs an AND operation of a 16-bit and an 8-bit variable
reg16	*INT*	2	`moviI rInt_3, #const`	Moves a constant value to a 16-bit variable
mem	*ADD(mem, reg8))*	2	`addGB @(rByte_3), rByte_5`	Adds and stores an 8-bit variable to a memory location
stmt	*SUB(reg32, reg16)*	2	`subLI rLong_2, rInt_3`	Subtracts a 16-bit variable from a 32-bit variable
stmt	*SET(INDIR(INT), XOR(INDIR(INT), reg16))*	2	`xorIOI #ioPort, rInt_3`	Performs an EX-OR of an I/O port value and a 16-bit variable
stmt	*SET (INDIR(REGISTER), AND(mem,INT))*	2	`andiG @(rByte_5), #const`	Performs an AND operation of the memory contents to a constant value and stores it to the same memory location
mem	*SET (GLOBAL, AND(GLOBAL,INT))*	2	`andiG @(symbol), #const`	Performs an AND operation of the memory contents pointed by a symbol to a constant value

where rByte_2, rByte_3 and rByte_5 are 8-bit pseudoregisters, rInt_5 and rInt_3 are 16-bit pseudoregisters and rLong_2 is a 32-bit pseudoregister.

Figure 4.11: Some grammar rules used by our customized BURG-generator.

micro-architecture of the datapath. A partial set of such "customized" rules, present in our BURG-generator and supported by a hardware micro-task datapath, is shown in the form of a table in Figure 4.11. For the sake of elaboration, we discuss the BURG rule given in Row 3.

The replacement symbol (S) in this rule is `stmt` *nonterminal* while the pattern (P) is `SET(INDIR(INT),OR(INDIR(INT),reg8))` which represents a combination of two operations present in a CDFG:

- an OR operation between the contents of a pointed location and an 8-bit variable.
- an assignment operation of the result achieved in previous operation to the pointed location.

The effective cost (C) of selecting this rule is 2 (as it takes 2 clock-cycles to implement this pattern in hardware) and the machine instruction generated through the action (A) performed is `orIOB #ioPort, rByte_2` where `rByte_2` is an 8-bit pseudoregister.

Following the classical BURG-based instruction selection approach, the tool generates a pattern matcher that contains all the patterns defined by the input grammar to the BURG-generator. The tool is integrated in Eclipse [133] and benefits from the code generation frameworks (such as Java Emitter Template (JET) editor [135]). Using the JET template defining the BURG rules, the tool generates JAVA code for the pattern matcher. Using this matcher, the instruction covering of the input CDFG IR is then a two-phase problem.

In first phase, that is a bottom-up traversal, all the nodes of the subject expression-tree are labeled with states that contain all the possible rules that can be used to reduce

this node to a possible *nonterminal*. In the second phase, that is a top-down traversal of the subject tree, the least-cost covering of the expression-tree nodes is found that reduce these node to `stmt` *nonterminal* and generates the resultant low-level assembly-like IR having *typed-pseudoregisters* (as shown in Figure 4.11, Column 4).

We would also like to point out that our instruction selection can be easily extended and it is highly customizable as many efficient rules that exploit the features of the underlying datapath can be easily added to our proposed grammar. During the course of the development of our design-flow, we experimented with *register-register* based, *register-memory* based and *memory-memory* based instruction patterns and finally we achieved an instruction selector that can easily exploit the micro-task datapath with *register-register*, *register-memory*, *memory-register*, *io-memory*, *io-register* and several similar instruction patterns.

4.2.3 Bitwidth adaptation

The next stage in our design-flow involves a wordlength conversion step in which the low-level assembly-like IR having different operations on 16-bit, 32-bit or 64-bit operands (e.g. typed-pseudoregisters) are transformed into either sequential byte-level (8-bit) or sequential short-level (16-bit) operations in order to match the characteristics of the underlying micro-task datapath. A datapath of smaller bitwidth in a hardware micro-task is more efficient from the point of view of area and power dissipation. A smaller bitwidth datapath contains relatively smaller busses (in width) and glue logic like multiplexers. This results in a smaller dynamic and static power consumption and of course, a smaller foot-print for the datapath. The price to be paid is a reduction in performance as, for example, it takes twice as long to perform a 16-bit arithmetic operation on an 8-bit datapath than on a 16-bit datapath. However, since the required operating frequency is not that high for WSN applications, we can pay this price for a reduction in silicon area and power consumption. Currently we can generate either an 8-bit and a 16-bit datapaths. The details of the design space exploration and comparison of power/energy/area consumptions of 8-bit and 16-bit datapath micro-tasks are discussed in Section 6.4.

The bitwidth adaptation phase has its own particular pit-falls that are to be treated with care while performing this step. Some of the issues involved are discussed in the following paragraphs.

- ***Propagation of carry***: The propagation of carry/borrow while replacing a 32-bit addition/subtraction to a sequence of four 8-bit add/sub instructions is to be handled carefully. In our case, we replaced a 32-bit `addLL` instruction with one 8-bit `add` and three 8-bit `addc` instructions. Similar (but modified accordingly) approach was used for 32-bit to 16-bit and 16-bit to 8-bit instructions too.

- ***Proper incrementation of pseudoregister-number***: Similarly, one has to take care about the pseudoregister-number being generated for 32-bit, 16-bit and 8-bit pseudo-registers. To avoid the over-writing of a pseudoregister by mistake

during bitwidth adaptation, we have to properly increment the pseudoregister-number according to its type during instruction selection and assembly-like IR generation.

- ***Conversion of comparison operation***: The `compare` operation can also be a difficult problem while performing bitwidth adaptation as comparison of two 32-bit operations cannot be directly replaced by a sequence of four 8-bit comparisons as the `compare` operation is accompanied by a `branch` operation. To explain the issue with the help of a simple example, we consider an instruction `cmpLL rLong_3, rLong_9` where `rLong_3` and `rLong_9` are 32-bit pseudoregister. If we want to convert this 32-bit instruction to its equivalent sequence of 16-bit instructions, we need to replace each of two 32-bit pseudoregisters present in the instruction by two 16-bit registers (as shown in Figure 4.12, Case A). The contents of `rLong_9` are supposed to be in two adjacent locations in register-file i.e. `r10` and `r9` while the contents of `rLong_3` are at `r4` and `r3`. Then, we perform the `compare` operation on `r4` and `r10` that contain the higher 16-bits and depending on the result, we perform the `branch` operation and then, we add the `compare` operation of the lower 16-bits using `r3` and `r9` and add the proper `branch` operation afterward. Moreover, the conversion of the `compare` operation involving operands of different bitwidth has to be handled differently. The procedure used in our bitwidth adaptation phase is shown in Figure 4.12(Case B) where we are comparing a 32-bit operand (`rLong_3`) with a 16-bit operand (`rInt_9`). We, at first, compare the higher 16-bits (i.e. the contents of `r4`) with 0. Then the `branch` operation is appended that will be performed depending upon the result of `compare` and then the `compare` operation having the lower 16-bits of first operand (i.e. the contents of `r3`) and the second operand (`r9`) is performed and its corresponding `branch` operation is added afterward.

  Similar transformations are also used during the bitwidth adaptation of the `Shift Left` and `Shift Right` operations. For instance, a 32-bit `shliL` instruction is replaced with a sequence of one 8-bit `shl` and three 8-bit `shlc` instructions to propagate the carry in sequential `shift` operations.

4.2.4 Register allocation

The only explicit resource binding operation performed in our design-flow is register allocation. In this step the temporary pseudoregisters used during the instruction-selection phase are replaced by physical registers present in the register-file of our micro-task datapath. We used a similar register allocation approach as was used by Chaitin et al. [18]. We used a linear approximation based algorithm to implement the graph coloring algorithm.

However, we do not implement the register-spilling step during the register allocation at the moment. Since the number of registers available in a register-file of our

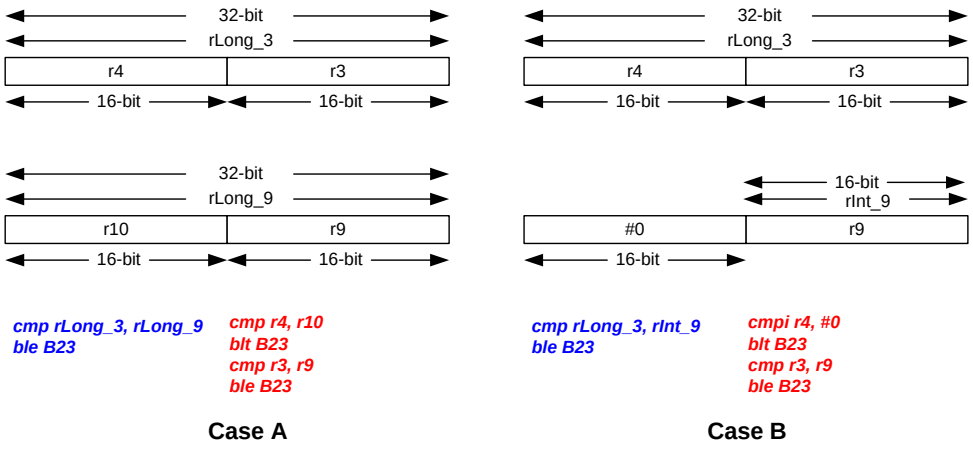


Figure 4.12: Bitwidth adaptation of the `compare` and `branch` instructions.

hardware micro-task is customizable, we gradually increase the number of physical registers during register allocation if the previous value results in an allocation failure.

It turns out that the number of registers required for allocation in WSN-related applications remains quite small (ranging from 2 to 8, as will be seen in Chapter 6). Nevertheless, register spilling may be implemented, in future, to generate a hardware micro-task datapath with a maximum fixed size of register file and the trade-offs would be studied for power/area/energy consumption comparison between a larger register file and a larger external data memory carrying the data of the spilled registers.

4.2.5 Hardware generation

The machine-specific IR obtained after register allocation is then processed through the FSM and datapath generation tools to generate an RTL-level IR of the hardware micro-task. The main reason for using this RTL-level IR is its reusability for code generation facilities. Once the RTL-level IR is generated, it can be easily retargeted to generate different back-end descriptions, such as VHDL, SystemC and C (for C-based behavioral simulator of the hardware micro-task that is discussed later in Section 4.2.5.3).

There are two parts of the hardware micro-task RTL-level IR (i) a semi-custom datapath (ii) an FSM, in which each machine-specific instruction, present in the assembly-like IR, is mapped to a sequence of micro-code (i.e. FSM states) used to control the micro-task datapath. The hardware generation phase for each of these two components is discussed in the following sections.

4.2.5.1 Datapath generation

We have developed, using the "Eclipse Modeling Framework (EMF)" [134], an RTL-level template for a generic datapath that can be used to construct an RTL-level IR of any circuit such as an FFT, DCT or FIR. This template contains the RTL models for wires,

I/O ports, logic and arithmetic operators, and control-flow and data-flow multiplexers. Similarly, we have also defined RTL libraries for different storage components such as dual-port RAM, single-port RAM, dual-port ROM, single-port ROM, shift register and RS flip-flop in this generic datapath template.

From the low-level assembly-like IR (generated after register allocation) we extract the required information for the generation of a customized datapath using the RTL-level datapath template mentioned-above. We tune different parameters of the components present in the generic datapath template to the application at hand. For example, the constants present in the input assembly-like IR are used to determine the size and the contents of the single-port immediate ROM to be generated. The operations are used to extract the information about the minimum-required functionality of the resultant ALU block while the maximum value of register-number present in the input IR is used to determine the size of the output register file. As a result, a customized datapath for the hardware micro-task is generated following the generic template shown in Figure 4.6.

4.2.5.2 FSM generation

As far as the generation of micro-coded FSM is concerned, we developed a Domain Specific Language (DSL), called *FSM-Sequencer*, that is based on an EMF-based tool Xtext [136] and can be used to describe the control-flows. A control-flow described in the *FSM-Sequencer* has inputs/outputs and control-sequences, and can be translated into a low-level RTL description of equivalent FSM.

Figure 4.13 shows a control-flow described using *FSM-Sequencer* where inputs and outputs (having default values) are contained in the header block. As far as the body block is concerned, each line represent a control-sequence where some outputs are being assigned. An example of such control-sequences is `set rdDstW=true, rdDstAdr=2` (shown in Figure 4.13) where `rdDstW` and `rdDstAdr` are the outputs being assigned. These control-sequences can be translated into corresponding FSM-states as it is shown in Figure 4.14 (a). In addition, we do not need to explicitly define transitions between consecutive control-sequences in the DSL. A transition takes place either to the following control-sequence in the code when a simple "`;`" is used or to another basic-block if *goto* construct is used. These transitions are implicitly defined in the syntax of *FSM-Sequencer*.

However, more complex transitions such as conditional branches can be described using *if*, *else* and *goto* constructs that can be translated into conditional transitions in an RTL-level FSM description. An example of such conditional branch (extracted from Figure 4.13) is

```
if (!lt_GT0 & lt_GT1 & lt_GT2) { goto B12; }
                    else { goto B4; }
```

where the control-flow goes to either label `B12` or `B4` depending upon the predicate value. Figure 4.14 (b) shows the corresponding FSM description that results in two conditional FSM-transitions. In short, *FSM-Sequencer* looks much similar to assembly language and a control-flow described in *FSM-Sequencer* can be easily translated into an RTL-level FSM description.

```
sequencer sendBeacon_fsm is (
	/*****************************************************************/
	/* Header contains the inputs and outputs to the control-flow
	   where output can be initialized with default values */
	input sendBeacon_enable : boolean;
	input lt_GT0 : boolean; input lt_GT1 : boolean; input lt_GT2 : boolean;
	output rdSrcAdr : int<2>:=0; output rdDstAdr : int<2>:=0;
	output rdDstW : boolean := false; output gVMASel1 : boolean := false;
	output gVMDSel : boolean := false; output gVMDirAdr : int<4>:=0;
	output gVMW : boolean := false
	/*****************************************************************/
)
begin
B4:
	/* Every line represent a control-sequence with outputs being assigned
	   ";" marks the transition to next consecutive control-sequence */
	set rfInSel0=true, rfInSel1=false, immRAdr=0;
	set rdDstW=true, rdDstAdr=2;
	set aluOp0=false, aluOp1=false, aluOp2=false, op1Sel0=false, op1Sel1=false, op2Sel0=false;
	set rfInSel0=true, rfInSel1=true, rdDstW=true, rdDstAdr=1;
	set aluOp0=false, aluOp1=true, aluOp2=false, op1Sel0=false, op1Sel1=false, op2Sel0=false;
	set rfInSel0=true, rfInSel1=true, rdDstW=true, rdDstAdr=0;

	/* Conditional branches are represented using if, else and goto constructs,
	   Unlike high-level languages, the branch predicate evaluation does not
	   consume any clock-cycle, so we have to add a nop operation to wait for
	   a clock-cycle in order to have correct values for the predicate variables*/

	if (!lt_GT0 & lt_GT1 & lt_GT2) {
		nop goto B12;
	} else {
		nop goto B4;
	}
B12:
	set rfInSel0=true, rfInSel1=false, immRAdr=6;
	set rdDstW=true, rdDstAdr=1;
	set ioDSel0=true, ioDSel1=false, ioPortW=true, ioPortAdr=4, rdSrcAdr=1;

	set sendBeacon_event0=true;
	set sendBeacon_event1=true, goto B4;
end
```

Figure 4.13: Description of a control-flow using *FSM-Sequencer* DSL.

There are several back-end passes developed for the generation of different output representations of the control-flows described in *FSM-Sequencer*. For example, an RTL-level IR description for the FSM can be generated. This RTL-level IR for the FSMs can be integrated in the RTL-level IR of datapath to complete the RTL-level IR of the complete hardware micro-task. Similarly, we can also generate the C and SytemC descriptions for the control-flows written in *FSM-Sequencer*.

We created a back-end pass that takes the low-level assembly-like IR generated after the register allocation and generates a control-flow description in *FSM-Sequencer*. Each control-sequence of this control-flow carries the micro-coded control signals for the underlying micro-task datapath. Above-mentioned code generation tools are then used to generate the corresponding RTL-level IR for different FSM components such as states, transitions, inputs and output ports.

4.2.5.3 Code generation

The *Eclipse Modeling Framework (EMF)* provides code-generation facilities that can be used to write back-end templates for different output codes (such as VHDL and SystemC). We use these facilities to write the VHDL templates for all the library components present in the RTL-level generic datapath template such as the FSM, the dual-port RAM, the single-port ROM, the ALU block and the multiplexers etc.

In the last step of our design-flow, using these VHDL templates for datapath and FSM components, we generate VHDL description for the hardware micro-tasks described in the RTL-level IR.

Similarly, we used the code generation facilities to write the C-based simulator templates for the assembly-like IR of the hardware micro-task code to perform the cycle-accurate behavioral simulation of the micro-tasks. A C-based behavioral simulator is used to simulate the behavior of a circuit in C. It can be used to perform debugging and behavioral validation of the circuit. There exist C-based Instruction-Set Simulators (ISSs) for different low-power MCUs (such as the MSP430 and the AVR) and some of them are also integrated in the WSN-node and network simulators (such as WSim [62] and WSNet [63]). This cycle-accurate micro-task behavioral simulator could serve for early validation and debugging of the hardware micro-task synthesis design-flow. Moreover, the generated simulators would also be integrated (in the future) in *WSim* and *WSNet* for a complete system-level and network-level validation.

4.2.6 Comparison to traditional design-flows of ASIP and HLS

Our design-flow is a hybrid of traditional ASIP and HLS design-flows. In traditional HLS, there is no notion of instruction pattern selection and mapping but an FSM-based controller is directly generated after analyzing the application code. On the contrary, our tool based on a retargetable compiler performs the instruction selection and mapping similar to a retargetable ASIP-like compiler infrastructure. Moreover, in traditional HLS, the tool is provided with a library of available hardware components and it generates itself a datapath based on an iterative approach. In contrast, similar

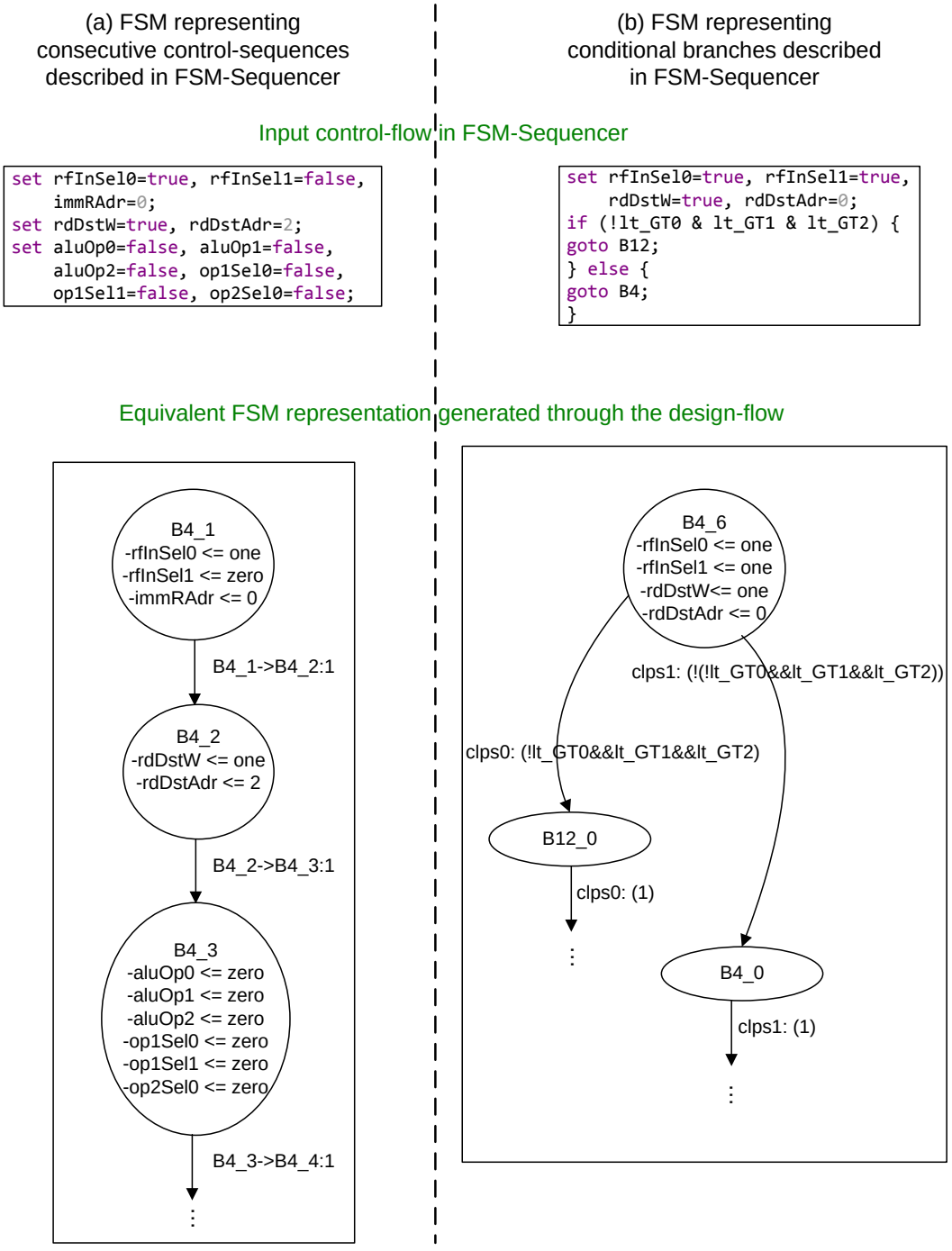


Figure 4.14: FSM representations generated through our tool for equivalent control-flows described in *FSM-Sequencer* DSL.

	HLS flow	**ASIP synthesis flow**	**LoMiTa design-flow**
Datapath (DP) selection	Iterative (mostly)	User-guided (ISA based)	User-guided (ISA based)
Instruction selection	No	Yes	Yes
Hardware generation	FSM + DP	ISA-based processor	FSM + DP
Application domain	Compute-intensive	Compute-intensive	Control-oriented

Table 4.1: Comparison of major features of the proposed approach to the existing ones.

to "UGH" approach proposed by Augé et al. [10] that uses a datapath-level abstraction to help coprocessor generation, our tool is provided with a higher-level (instruction-set level) abstraction to help the hardware micro-task synthesis. In addition, our approach may seem similar to the processor specialization of Gorjiara et al. [11], however, our main goal is to minimize the silicon footprint of the resulting hardware micro-task, improving performance is only a secondary objective.

As far as comparison to ASIP design-flow is concerned, our tool performs the instruction selection and mapping but does not follow the classical object-code and processor synthesis path. Instead it generates a micro-coded FSM that is used to control the micro-task datapath. Table 4.1 compares some of the major features of the three design-flows.

The hardware micro-tasks generated through our design-flow show good results in terms of not only power and energy consumption but area cost as well. The details of experimental setup, application benchmarks and a practical case-study are presented in Chapter 6. However, the next section provides some results for a control-oriented task processed through our micro-task generation tool.

4.3 An illustrative example of micro-task synthesis

As it has been discussed by Raval et al. [119], major part of a WSN node workload consists of communicating with the RF transceiver for data exchange. This data-exchange is normally performed between the on-chip MCU and the RF transceiver through SPI-link.

In this section, we present a small example of a C-code to explain the working of our design-flow. This C-code is a function called `sendBeacon()` that is used by a receiver WSN node to send a beacon-frame to a transmitter node through its RF transceiver. A part of this particular C-code is shown in Figure 4.15. Since this application code contains the instructions that directly communicate with the I/O ports of an RF transceiver, we will take this opportunity to show the specialized instruction selection of our tool to select the I/O-operand based instructions. We have highlighted the portion of C-code that will be concentrated upon throughout this example.

The input C-code is processed through the GeCoS front-end and an IR in the form of a CDFG is generated (as shown in Figure 4.16). This CDFG (IR) is then processed through *instruction selection*, *bitwidth adaptation* and *register allocation* phases, and a low-level assembly-like IR is generated. Figure 4.17 shows the IR that corresponds to

```
for (i = 0; i < 8; ++i) {
              (*(volatile unsigned char*)0x04) = sentFrame[i];
              while (((*(volatile unsigned char*)0x02) & 0x01) == 0);
       }
       /*************************************************/
       // The part of the C-code under study that communicates
       // with the I/O-ports addressed by 0x01 and 0x04
       (*(volatile unsigned char*)0x01) |= 0x01  ;
       (*(volatile unsigned char*)0x01) &= ~(0x01);
       (*(volatile unsigned char*)0x04) = 0x03;

       /*************************************************/
       while (((*(volatile unsigned char*)0x02) & 0x01) == 0);

       (*(volatile unsigned char*)0x01) |= 0x01;
       do
       {
              (*(volatile unsigned char*)0x01) &= ~(0x01);
              (*(volatile unsigned char*)0x04) = 0x00;
              while (((*(volatile unsigned char*)0x02) & 0x01) == 0);

              spiStatusByte = (*(volatile unsigned char*)0x03);
              (*(volatile unsigned char*)0x01) |= 0x01;
       }
```

Figure 4.15: A portion C function `sendBeacon()` under study.

the C-code under study. Here, we can clearly see the I/O-operand based specialized instructions such as `orIOi` and `movIOi` being generated.

This IR is then processed through the hardware generation stage of our tool and it is converted to its corresponding RT-level EMF-based model of a hardware micro-task that is further processed through the code-generation step to generate the synthesizable VHDL description.

4.3.1 Resultant dynamic power and energy savings

We present below a comparison of power and energy consumption of `sendBeacon()` task when implemented in hardware (generated through our tool) and in software on a low-power MSP430F21x2.

The original C-code for `sendBeacon()` when run on the MSP430-core consumes on average 278 nJ for one execution while working at 16 MHz. The equivalent hardware micro-task generated through our tool, for 130 nm technology, consumes only 1.4 nJ of energy for one execution while working at same frequency. As far as, the power consumption is concerned, the MSP430-core consumes around 8.8 mW when working at 16 MHz while the equivalent hardware micro-task consumes approximately 33 μW. As a result, we get a 264 x power while 198.5 x energy gain respectively while executing the same control task in hardware generated through our tool. More details of our experimental results are presented in Chapter 6).

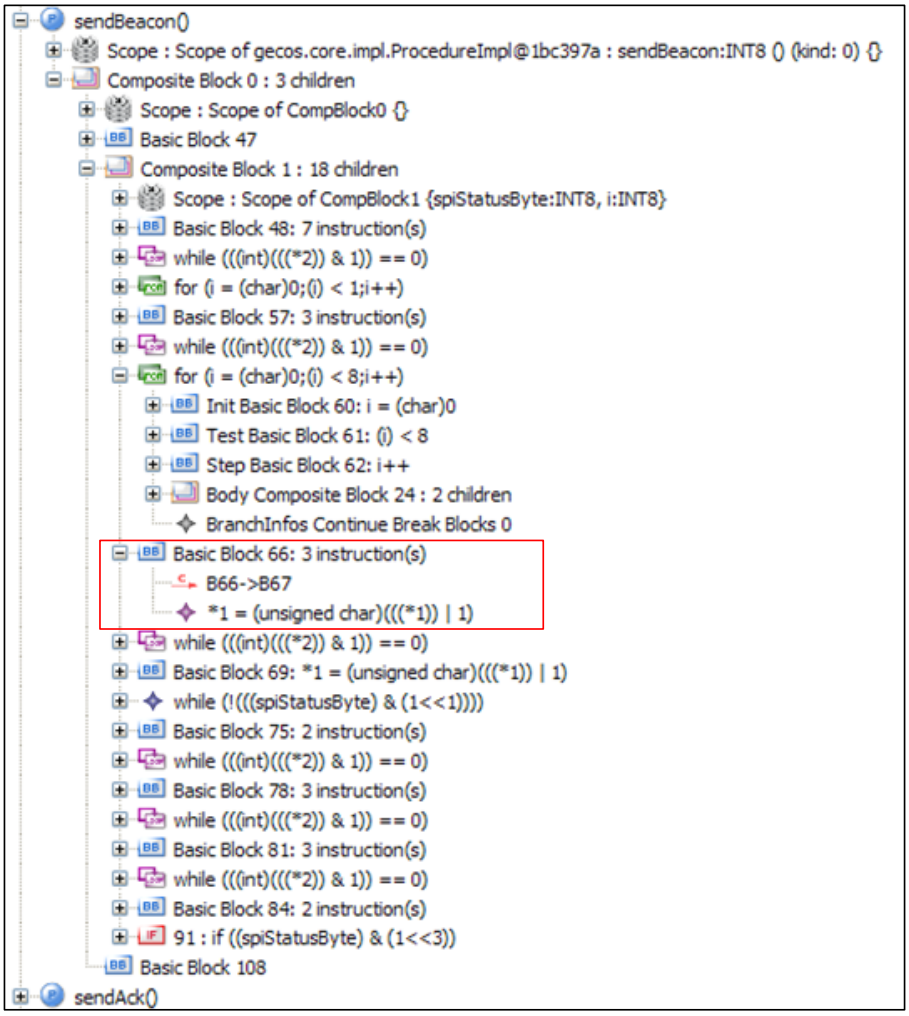


Figure 4.16: CDFG representation of the C-code under study.

```
B27:
       jmp    B26
B23:
       addi   %r0, #1
       jmp    B22

/*************************************************************/
// B30 corresponds to the highlighted part C-code in Figure 4.15
 B30:
       orIOi  %io0(#1), #1
       andIOi %io0(#1), #0
       movIOi %io0(#4), #3
/*************************************************************/

B31:
       movi   %r1, #2
       movi   %r0, #1
       and    @(%r1), %r0
       movi   %r0, #0
       cmp    @(%r1), %r0
       bne    B34
B32:
       jmp    B31
B34:
       orIOi  %io0(#1), #1
B36:
       movIOi %io0(#1), #0
       movIOi %io0(#4), #0
```

Figure 4.17: Machine-specific intermediate representation of the C-code under study.

As we discussed in Chapter 1, our work consists in a complete design-flow for the generation of ultra low-power WSN-node controllers. On one hand, our design-flow consists in a C to RTL VHDL generation of hardware micro-tasks, while on the other hand, the second contribution of our work consists in developing a design-flow for the generation of a hardware System Monitor (SM) that is responsible for the scheduling of these hardware micro-tasks.

Next chapter presents the details about the notion of our system-level execution model and its features, the corresponding existing work as well as the design-flow for the synthesis of the SM responsible of implementing this system-level execution model.

Chapter 5

Proposed system model and design-flow for SM synthesis

This chapter presents the system-level execution model of the computational and control subsystem of a WSN node based on our approach. We start this chapter by briefly discussing the existing execution paradigms in embedded systems as our target domain (WSN systems) is a sub-category of embedded systems. We then continue with describing the basic system-level view of proposed micro-task-based WSN node. Since the current WSN node implementations (such as Mica, Mica2, ScatterWeb etc.) are built on conventional MCUs and they use WSN-specific OS for system-level task management, the second part of this chapter details the related work done in the domain of WSN-specific OS. The chapter continues onward with the features of our proposed execution model and the notion of a hardware System Monitor (SM) that is used to perform the task- and power- management of a micro-task-based WSN node. Finally, we conclude this chapter with the details of our design-flow developed for the generation of an RTL description of the SM from a high-level system description (as shown in Figure 5.1) and some experimental results about the power and area consumption of the SM generated for the example presented in Section 1.3.2.

5.1 Basic execution paradigms in a WSN node

The traditional tasks associated to an Operating System (OS) are the control and protection of resource-access (including support for I/O), and management of resource allocation to different users. Moreover, the support for concurrent execution of several processes and their inter-communication is also considered as a job of the OS. These functionalities are, however, only partially required in an embedded system as code execution is much more restricted and usually more tightly synchronized than in a general-purpose system. Moreover, as the description of the microcontrollers has shown, in Section 2.5, these systems simply do not have the required resources to support a full-blown OS.

Similarly, an OS or an execution paradigm for WSN nodes should support the

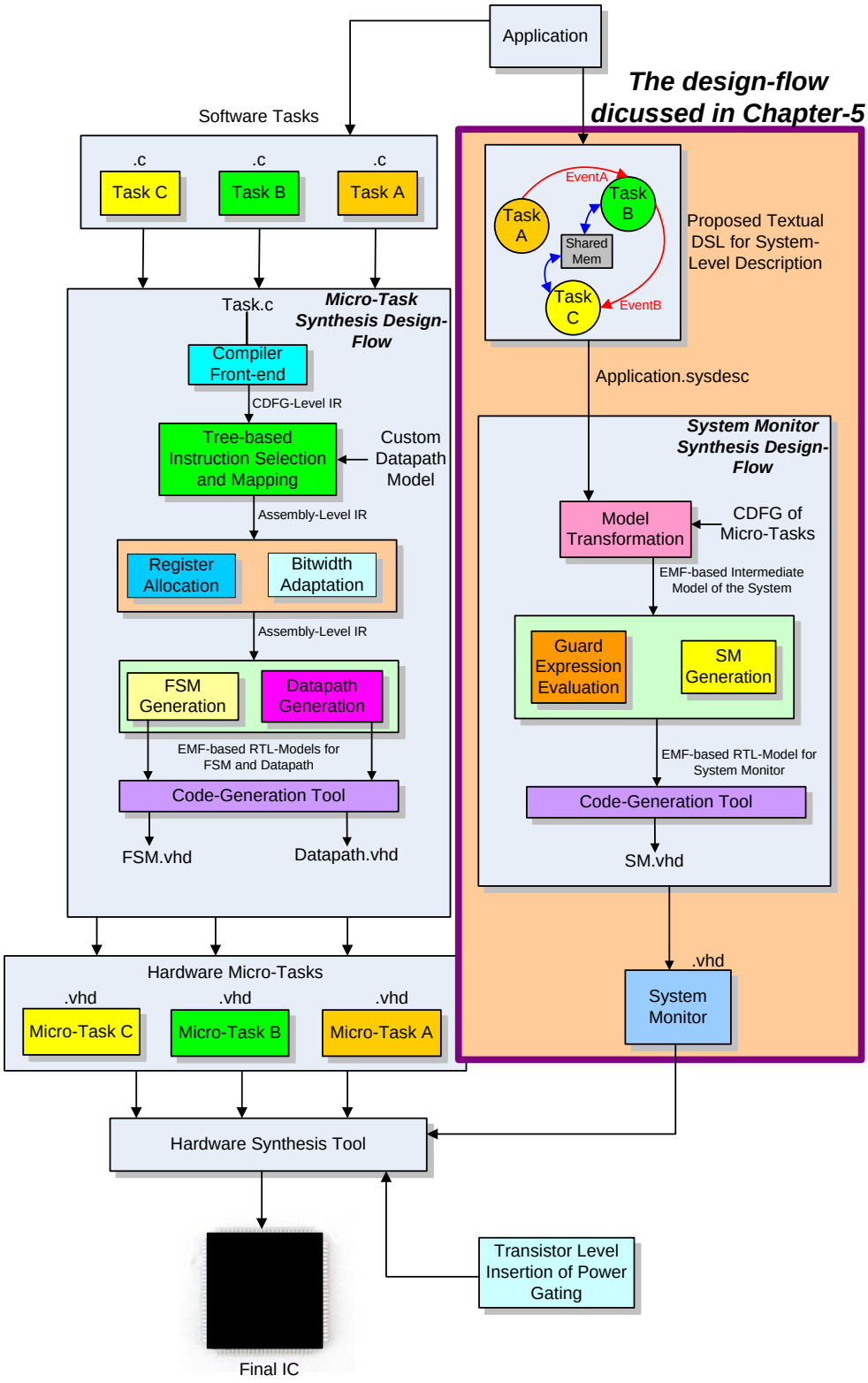


Figure 5.1: Design-flow for hardware system monitor generation.

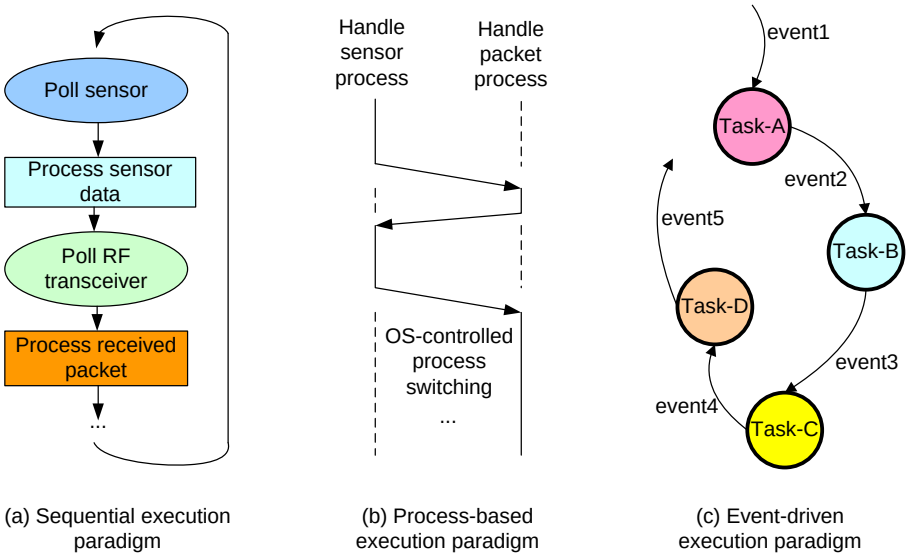


Figure 5.2: Different execution paradigms for a WSN node system.

specific requirement of these systems. In particular, energy-efficient execution requires support for energy management. Similarly, external components (e.g. sensors, radio transceiver, or timers) should be handled easily and efficiently, specially information that is available asynchronously (at any arbitrary time-instance) must be handled. There are three possible approaches to handle such tasks in conventional embedded systems [70]. In the following discussion, we briefly explore these approaches.

5.1.1 Sequential approach

The first and the simplest approach to handle tasks in a system is the sequential approach. For example, a system could poll a sensor to decide whether some data is available and process the data right away, then poll the radio to check whether a packet is available, and process the packet, and so on (as shown in Figure 5.2 (a)). However, such a simple model risks of missing data while a packet is processed or missing a packet when sensor information is processed. This risk is particularly large if the sensor-data processing or incoming-packet processing takes substantial amount of time, which can easily be the case. Hence, a simple, sequential approach is clearly insufficient.

5.1.2 Process-based approach

Most modern, general-purpose OS support concurrent (quasi-parallel) execution of multiple processes on a single processor. Hence, such a process-based approach would be the first candidate to support concurrency in a sensor node as well. The approach is shown in Figure 5.2 (b) where different processes are run on a single resource (an MCU) and their switching is controlled by an OS. While indeed this approach works in principle, mapping such an execution model of concurrent processes to a sensor node

shows however some inherent problems. For example, often the tasks to be executed in a WSN node are smaller w.r.t. the overhead incurred for switching between tasks. Also, each process requires its own stack space in memory, which does not fit properly with the stringent memory constraints of sensor nodes.

5.1.3 Event-driven approach

For these reasons, a relatively different execution model seems more appropriate. The idea is to take into account the reactive nature of a WSN node. In this case, the system essentially waits for any event to happen where an event can be the availability of data from a sensor, the arrival of a packet, or the expiration of a timer and reacts to these events by performing certain tasks. This approach is called event-driven execution paradigm. Since all the events are destined for small tasks to be performed, the whole WSN application can be presented in the form of Task Flow Graphs (TFGs). Such an approach is presented in Figure 5.2 (c) where different application and control tasks, present in a system, communicate with each other through events.

5.2 System-level execution model

We use event-driven paradigm as the system-level execution model in our approach. To give an example, Figure 5.3 shows the TFGs of a lamp switching application, where a transmitting node demands a receiving node to switch on/off its lamp if a button is pressed at transmitter end. Figure 5.3 (a) shows the TFG for receive mode when a node waits for a signal from the transmitter and switches the lamp, whereas Figure 5.3 (b) presents the TFG for transmit node where a node waits for push-button event and sends a signal to the receiver to switch the lamp. This application involves several tasks such as data transmission, data reception, wait for acknowledgment, and timer, push-button and lamp switching APIs.

All these control-oriented tasks are spread across different layers of the communication stack and involve further sub-tasks associated to them. For instance, beacon and data packets transmission and reception involve physical layer functions that exchange data between I/O peripherals of the MCU and the RF transceiver using SPI-protocol. The control-flow itself follows a simplified version of RICER, a low-power MAC protocol [84]. In typical WSN node, such tasks are handled by an MCU and corresponding OS that provides support for multi-tasking features.

In our proposed approach, all of the tasks present in such TFGs are executed on dedicated hardware resources (i.e. the hardware micro-tasks). Figure 5.4 presents a system-level view of a WSN node platform based on our proposed approach. Such a system consists of:

- A set of power-gated hardware micro-tasks accessing shared resources (e.g. peripherals (RF, sensor) and memories (gated/non-gated)). Each of these hardware micro-tasks is able to perform a specific task such as temperature sensing, processing data, sending data to SPI-interface etc.

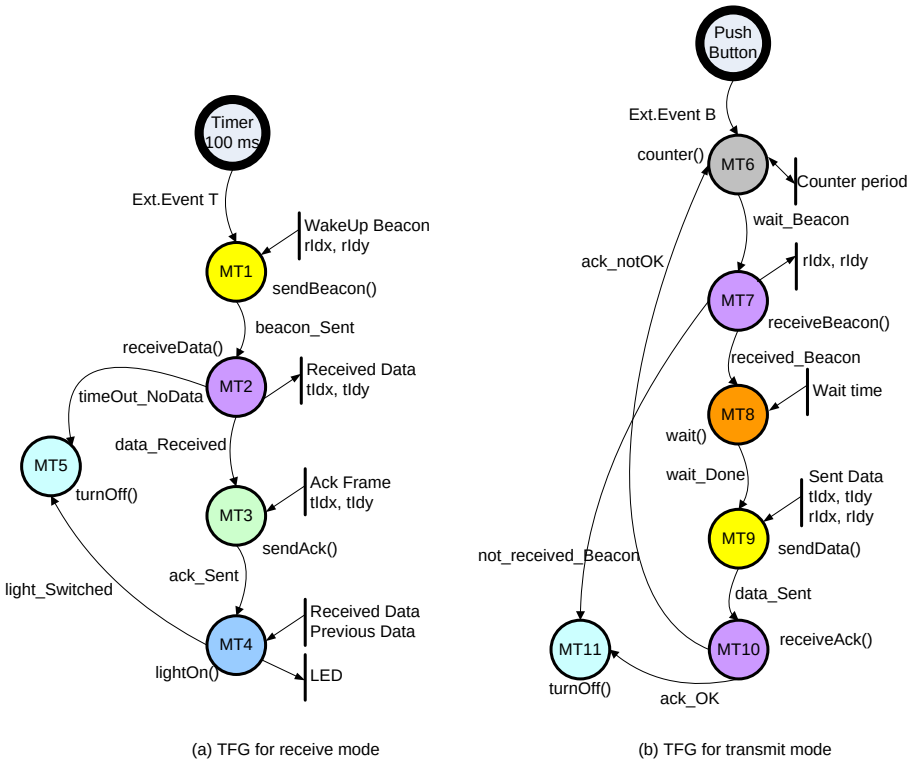


Figure 5.3: TFGs presenting the tasks running in a lamp switching application.

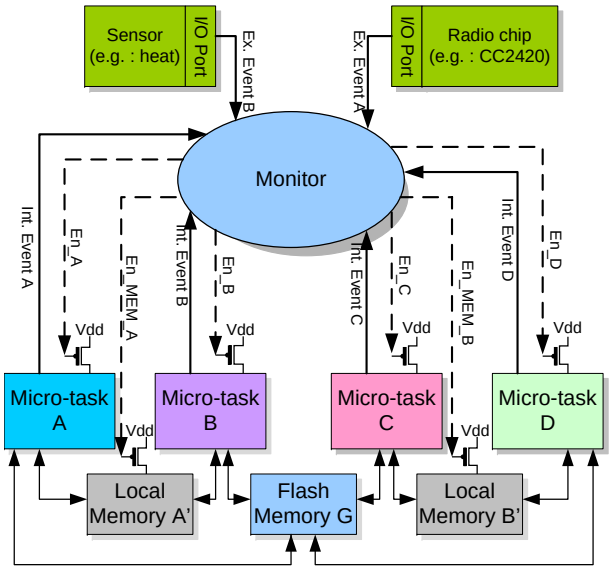


Figure 5.4: System-level view of a micro-task based WSN node architecture

- A hardware *System Monitor* (SM) that controls the execution of all the hardware micro-tasks. The SM is responsible for the turning-on/off of the hardware micro-tasks as well as the power-gated memories depending upon their usage. The detailed architecture, working of the SM and the design-flow for its synthesis are discussed later in this chapter.

- Event triggering peripherals (such as wake-up radio, timer etc.) that can send events to the SM.

- A set of memory resources that are used to store the global data shared among different micro-task. These memories can be power-gated or non-power-gated depending upon their usage.

Later in this chapter, we will provide the details of the specificities and limitations of our proposed system-level model and compare them to the features of standard OS-based WSN implementations. However, to be able to perform this comparison, we first need to take a closer look at the system-level view and management aspects of conventional OS-based WSN nodes. In typical embedded systems, concurrency, event and shared resource management are handled by a real-time embedded OS, which provides adequate constructs for the programmer (preemptive task scheduling, mutex, etc.). However, such full featured OSs cannot be implemented in WSN nodes because of the strong constraints on memory footprint. As a result, several light-weight operating systems have been developed by the WSN community that handle the power- and task-management in current WSN nodes. The next section summarizes the features of some of these WSN-specific OSs.

5.3 WSN-specific OS

Due to strong constraints on memory resources available on a WSN node, several WSN-specific OSs have evolved that are very light-weight in terms of memory requirement and can run on low-power embedded MCUs such those discussed in Section 2.5. This section outlines the characteristics of the most commonly used OS infrastructures targeted at WSN.

5.3.1 TinyOS

TinyOS [101] is one of the earliest and the most commonly used OS in WSN platforms. It is a flexible OS built from a set of reusable components that are assembled into an application-specific system. TinyOS supports an event-driven concurrency model and uses asynchronous events, and deferred computation called *tasks*. TinyOS is implemented in the NesC language [44], which supports the TinyOS component and concurrency model as well as cross-component optimization and compile-time race detection.

A TinyOS program is a graph of components where each component is an independent computational entity that exposes one or more interfaces. Components have three computational abstractions: *commands*, *events*, and *tasks*. *Commands* and *events*

are mechanisms for inter-component communication, while *tasks* are used to express intra-component concurrency and computation. A *command* is typically a request to a component to perform some service, such as initiating a sensor reading, while an *event* signals the completion of that service. *Commands* and *events* cannot block, rather a request for service is *split-phase* i.e. the *command* returns immediately and the *event* signals completion at a later time. *Tasks* may perform significant computation and follow a *run-to-completion* execution-model. This allows tasks to be much lighter-weight than threads. The standard TinyOS task scheduler uses a non-preemptive, FIFO scheduling policy. Since TinyOS uses a system of smaller components that are statically linked with the kernel to form a complete image of the system, so after linking, modifying the system is not possible. However, in order to provide run-time reprogramming for TinyOS, Levis et al. have developed Maté [81], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time.

5.3.2 Contiki

Contiki [29] is another WSN-specific OS proposed by Dunkels et al. It features an event-driven kernel where multi-threading is not present as an inherent feature but can be implemented as a library that is linked only with programs that explicitly require it. Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including the MSP430 and the Atmel AVR.

A running Contiki system consists of kernel, libraries, program loader, and a set of processes. A process may either be an application program or a *service*. A *service* implements a functionality used by more than one application processes. All processes, both application programs and services, can be dynamically replaced at run-time. Communication between processes always goes through the kernel. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A Contiki system is partitioned into two parts: the core and the loaded programs as shown in Figure 5.5. The partitioning is made at compile time and is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it is possible to use a special boot loader to overwrite or patch the core. Programs are loaded into the system by the program loader. The program loader may obtain the program binaries either by using the communication stack or by using directly attached storage such as Electrically Erasable Programmable ROM (EEPROM). Typically, programs to be loaded into the system are first stored in EEPROM before they are programmed into the code memory. So, the major benefit of Contiki OS over its competitors is its feature to dynamically load and unload an application program or service. But there is a question of the efficiency of this process as the number of nodes in a WSN can go up to 10,000, will it be possible to dynamically update all these nodes

in real-time?

Contiki uses a notion of *protothreads*. *Protothreads* are programming abstraction that provide a conditional blocking statement to simplify the event-driven programming for memory constrained systems. The operation takes the conditional statement and blocks the *protothread* until the statement evaluates to true. If the statement evaluates to true at the very first time, the *protothread* continues its execution without being interrupted; otherwise the *protothread* is blocked and condition is re-evaluated each time the *protothread* is invoked.

In Contiki, *protothreads* are stack-less i.e. local variable contents are lost whenever the scheduler switches from one *protothread* to another. Hence the programmer must take great care while using such variables inside his program. We have observed that programmers using Contiki in their WSN system usually avoid using local variables to reduce complications and due to the fact that they are not familiar with the concept. For instance, PowWow [64] is an open-source WSN platform, that uses Contiki as OS, where programmers have completely avoided the use of local variables.

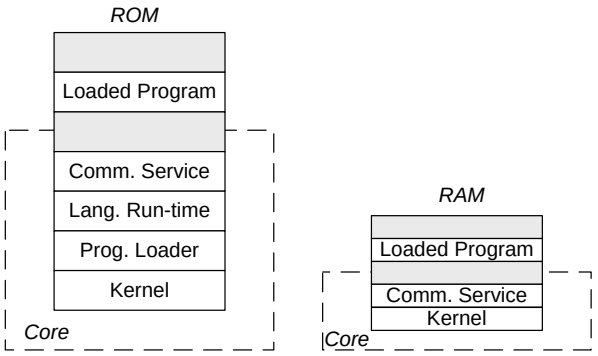


Figure 5.5: System overview of Contiki OS [29] (portioning into core and loaded programs).

5.3.3 MANTIS OS

The MANTIS Operating System (MOS) [14] uses a traditional multi-threaded approach. It offers a time-sliced approach in which an interleaved concurrency of multi-threading is used to prevent one long-lived task from blocking execution of a second time-sensitive task. However, a larger memory resource is needed for thread-management as task preemption requires that the complete stack of the preempted thread is to be saved. Average stack size of TinyOS is approximately 16 KB and it does not support multi-threading, while MOS dedicates 128 Bytes/thread having a multi-threading based kernel.

5.3.4 LIMOS

LIMOS [146] is a resource-aware, low-power-consuming and distributed real-time micro-kernel which is specially designed for the real-time applications. It uses the notions of *event* and *thread* to build a two-level system architecture. It also exploits a two-level scheduling policy: *non preemptive priority based* high-level scheduling for *events* and *preemptive priority-based* low-level scheduling for *threads*. The scheduling scheme is predictable and deterministic with respect to real-time applications. An *event* is the job-unit of LIMOS and a *thread* is an atomic unit of an *event*. *Threads* are the essential system units, each containing a block of independent program that has a particular function. A group of relative *threads* are organized into an *event*, each containing the *threads* that deal with a certain aspect of a system by cooperating with others. Therefore, LIMOS consists of a set of *events* and each event contains a number of relative *threads*.

LIMOS *events* follow a *run-to-completion* semantic without preempting one another i.e. *events* are non-preemptive and adopt a priority-based non-preemptive scheduling, such as Earliest-Deadline-First (EDF) algorithm. On the other hand, LIMOS *threads* run in parallel to implement an *event* by interacting with each other. Hence, *threads* are preemptive and thus each one needs a memory to store its *context* and other status information. There is a static priority-based preemptive scheduling scheme for *threads*. *Threads* are selected to run in order of priority and the selected *thread* can preempt any other lower priority *thread* at any execution point outside of critical. When the *threads* of an active *event* are running, the *threads* of other *events* are not eligible to obtain CPU resource. This allows *events* to run to completion. Since *threads* follow a static scheduling that is determined prior to run-time, the priorities of *threads* must be allocated carefully to avoid a deadlock situation.

5.3.5 SenOS

SenOS [73] is a totally different OS than all the above mentioned OSs due to the fact that it does not follow a conventional thread-based operation style. It deploys a state machine based programming model. A state machine has been recognized as a powerful modeling tool for reactive and control-driven embedded applications. Sensor network applications are one of those applications that can mechanize a sequence of actions, and handle discrete inputs and outputs differently according to its operating modes. Being in a state implies that a system reacts only to a predefined set of legal inputs, produces a subset of all possible outputs after performing a given function, and changes its state immediately in a mechanical way.

SenOS has four system-level components:

- an *event queue* that stores inputs in a FIFO order,
- a *state sequencer* that accepts an input from the event queue,
- a *callback function* library that defines output functions,

- a *re-loadable state transition table* that defines each valid state transition and its associated callback function. Each callback function should satisfy the *run-to-completion* semantics to maintain the instantaneous state transition semantics. This phenomenon is achieved by using a protected shared resource like mutex.

In SenOS, kernel and callback library are statically built and stored in flash ROM of a sensor node whereas the state transition table can be reloaded. The SenOS can handle multiple applications by means of multiple co-existing state transition tables and provide concurrency among applications by switching state transition tables. Each state transition table defines an application and during preemption, the kernel saves the present state of the current application, restores the state of the next application, and changes the current state transition table.

After presenting the existing related work about the most commonly used OS by the WSN community, in next section, we present the important features and notions used in our system-level execution model and compare them wherever possible to those of conventional OS-based WSN systems.

5.4 Features of our proposed execution model

First of all, we want to clarify that our goal (in this work) is not to propose a new model of computation for WSN computation subsystem. We rather see our proposed approach as a simple system-level execution model chosen so as to be a good match for what we think is a promising architectural solution for WSN nodes. The important notions or features proposed in our execution model are discussed in the following sections.

5.4.1 Events and commands

In our approach, we use *command* and *event* message structures between the SM and hardware micro-tasks similar to those of TinyOS. A *command* is an enable signal generated by the SM toward a hardware micro-task signaling the start of its operation. On the other hand, an *event* is a control signal generated by a hardware micro-task to the SM announcing the termination of its job.

Dotted lines in Figure 5.4, represent *command* signals driven by the SM to the micro-tasks and shared memories. Solid lines represent the *events* sent back to the SM. *Events* can be of two types:

- ***internal event***: an *event* that is generated by a hardware micro-task indicating its termination or preemption (in case of a sub-routine call).

- ***external event***: an *event* that is generated by an external peripheral that can serve as wake-up call from shut-down mode using a timer, arrival of a data packet at RF interface or a periodic value received at the sensor interface.

5.4.2 Concurrency management

Both TinyOS and Contiki allow the programmer to express concurrency in their applications through the use of specialized constructs such as *Protothreads* for Contiki and *Tasks* for TinyOS. One of the goals of such non-standard constructs is to help reducing context switching and task scheduling overhead.

In our approach, as we rely on several physically distinct hardware micro-tasks, we provide a natural support for concurrency and task-level parallelism, and do not have to pay for any context switching overhead. Similarly, scheduling does not have any execution time overhead, even if taking into account the extra silicon area required by the SM.

Another advantage of our approach lies in the fact that shared resources such as memory or I/O ports are much easier to handle than in a standard multi-processor architecture, thanks to *power-gating*.

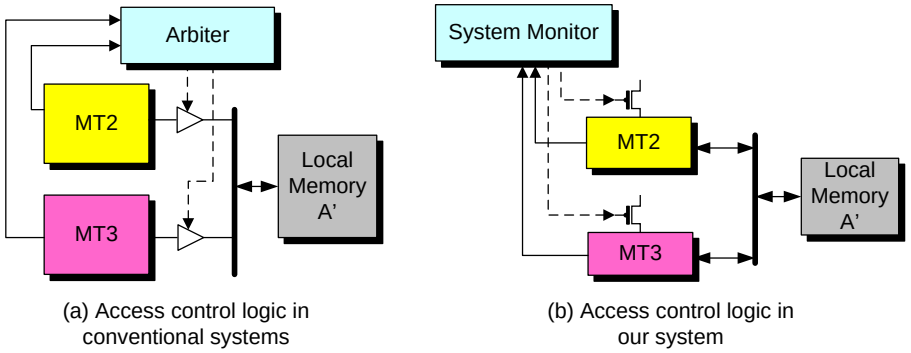


Figure 5.6: Access control simplicity of power-gated modules.

We introduce a resource constraint in our execution model that no two hardware micro-tasks sharing *write-access* to a shared resource can be active (powered-on) at the same time. Thanks to this constraint, we can save the typical extra tri-state (or multiplexer) logic used to share data/address bus lines, which results in savings both in terms of power and area. Figure 5.6 shows this concept while comparing the access control logic in our system compared to a conventional one.

5.4.3 Task hierarchy

In our execution model, we offer two ways for handling sub-routine calls made within the C-specification of a hardware micro-task.

The first one is straightforward and consists in inlining the sub-routine calls, this increases the task granularity and is acceptable for small sub-program calls. The second one is more complex and consists in generating a new hardware micro-task dedicated to the sub-program execution. In latter case, the parent (i.e. caller) micro-task invokes the child micro-task (through the SM). As the child micro-task is also power-gated, it only marginally contributes to the static power budget, while helping in maintaining a

high level of specialization within the parent task. The data communication between the parent and the child hardware micro-task is done through shared memory.

5.4.4 Memory management

There are small locally shared memories used by the hardware micro-tasks that can be power-gated once their corresponding micro-tasks are shut down. We must emphasize that a system-level model (see Section 5.6) is used to specify that, after the termination of a given task, which memory was being used by the task and this information is used to turn the shared memories off. This notion of small power-gated locally shared memories, instead of a large global one, will also contribute to the overall reduction in power consumption.

There is also a very small global memory (based on non-volatile flash technology) that is used to store the global data such as the node-ID, node-address, neighborhood table and if there is some potential data to be saved by the micro-tasks, in case of local memory shut-down. Since an *always-ON* memory can be critical from the point of view of static power dissipation, we use a non-volatile flash memory to store the needed data that can be turned-off in case of total node shut-down.

In this extremely low-power mode, only the SM will be powered-on while all the other components of the micro-task-based computational and control subsystem will be power-gated. In the next section, we explain the architecture and working of the SM being used for task- and power-management.

5.5 System monitor (SM)

All the features presented in previous section are made sure by a hardware scheduler present in our system, called the *System Monitor* (SM). The SM that is responsible for powering-on/off micro-tasks/memory blocks upon reception of an (or a combination of) event(s), is itself implemented as a simple combinational logic block to evaluate the guard conditions for micro-task activation and a set of 1-bit status registers carrying the state of events and commands signals until they are used by micro-tasks. Figure 5.7 provides a block diagram of the SM components for the case study example mentioned in Section 5.1.3.

In our execution model, we restrict ourselves to micro-tasks following a *run-to-completion* semantic, as in the case of TinyOS tasks. This ensures that a given micro-task will never reach a state in which it is activated (i.e. its powered-on) while not executing useful computation (i.e. blocked waiting for some event).

In addition, we make sure that at a given time instant there may not be two active tasks sharing a write-access to a same shared resource. This property is ensured (in a conservative way) by the SM which makes sure that, prior to activating a candidate hardware micro-task, there are no other active micro-tasks that *may* need write-access to a resource that *may* also be used by the candidate micro-task.

In the remaining part of this chapter, we will explain the design-methodology that

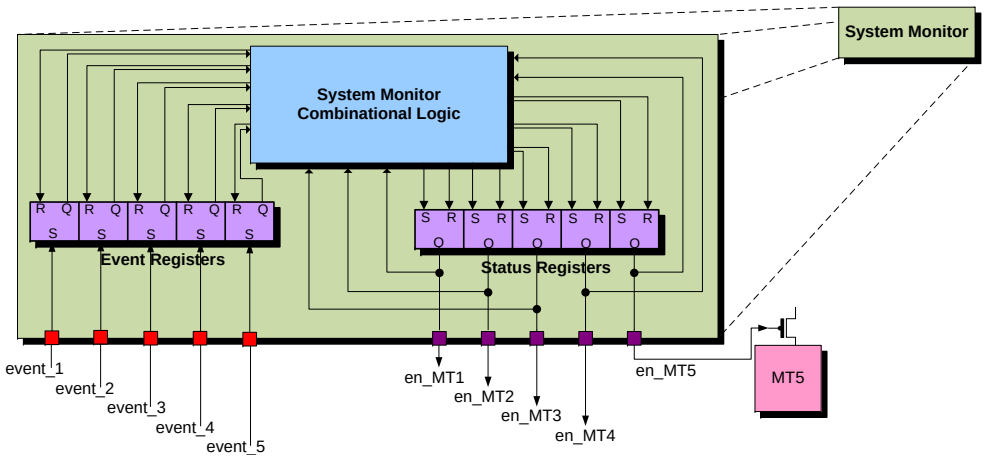


Figure 5.7: Block diagram of the *System Monitor* designed for the lamp switching example of Figure 5.3.

we have devised for the automatic synthesis of the SM from a high-level system description.

5.6 Design-flow for the SM generation

Figure 5.8 shows the second half of our tool, a design-flow developed for the generation of the SM that is used to control the system-level behavior of a micro-task-based WSN node. The basic steps involved in this design-flow are discussed in details in the following sections.

5.6.1 System specification

We developed a Domain Specific Language (DSL) that is used to specify the system-level execution model of a WSN node and its components e.g. micro-tasks, event, shared memories, peripherals, etc. This DSL is developed by using Xtext, the EMF-based MDE framework.

In order to write a system-level description of a micro-task-based node, we have to define all the external and internal events present in the system. Moreover, for each micro-task in the system, we specify its corresponding sub-program name, the event configuration (whether a single event or a logical combination of events) that is necessary for the micro-task activation and also the events produced by the micro-task at its termination. Similarly, for each global variable of the application-code, we specify which memory block (gated/non-gated) is used as storage component and the shared I/O ports that are used by the micro-task.

Figure 5.9 shows an example of the system-level description written in DSL. This description corresponds to a portion of the lamp-switching example shown in Figure 5.3,

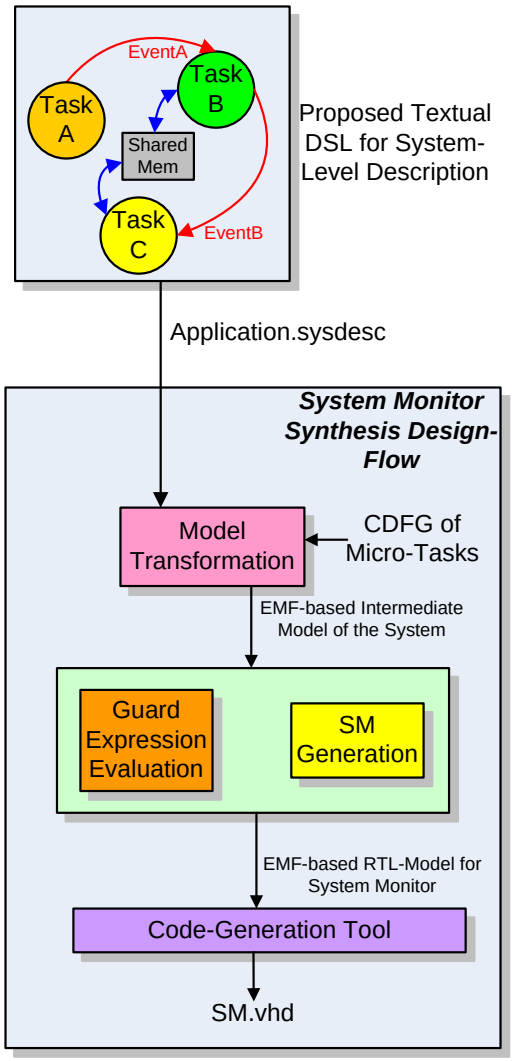


Figure 5.8: Design methodology for system monitor (SM) generation.

```
system send_receive_data {
	include "send_receive.gecos"   /* Link to CDFG IR of micro-task synthesis design-flow */
	/*****************************************************************************
	 *           Events existing in the system (both internal and external)      *
	*****************************************************************************/
	events { extPB, extET, beacon_Sent, data_Received,
	         ack_Sent, timeOut0, timeOut1, timeOut2, receiver_OFF, transmitter_OFF,
	         counter_Start, beacon_Received, data_Sent, ack_OK, ack_NOK, radio_OFF}
	/*****************************************************************************/
	/*****************************************************************************
	 *     Shared memories existing in the system (both gated and permanent)     *
	*****************************************************************************/
	memory memB [gated] {
		contains globals {neigh_IdX,neigh_IdY, receiveFrame, sentFrame, pushButtonStatus}
	};

	memory memC [permanent] {
		contains globals {my_IdX, my_IdY}
	};
	/*****************************************************************************/
	/*****************************************************************************
	 *                          Shared I/Os existing in the system               *
	*****************************************************************************/
	ioModule led {
		contains ports { port LED 8}
	};

	ioModule pushButton {
		contains ports { port PUSHBUTTON 7}
	};

	ioModule cc2420 {
		contains ports { port P2IN 0, port P5OUT 1, port U1TCTL 2, port U1RXBUF 3,
		                 port U1TXBUF 4, port URXIFG1 5, port IFG 6}
	};
	/*****************************************************************************/
	/*****************************************************************************
	 *                  Hardware micro-tasks existing in the system              *
	*****************************************************************************/
	microTask receiveData {
		activates With { beacon_Sent }
		produces { data_Received }
		reads ioModule { cc2420 }
		writes memory { memB }
	};

	microTask sendBeacon {
		activates With { extET }
		produces { beacon_Sent }
		writes ioModule { cc2420 }
		reads memory { memC }
	};
}
```

Figure 5.9: A snapshot of the system-level execution model, of the lamp-switching example shown in Figure 5.3, described using proposed DSL.

where the important notions of the execution model such as events, shared resources and micro-tasks are highlighted.

5.6.2 Model transformation

In the second step of the design-flow, the information provided by the system specification is processed through our tool and an IR of the system is generated that contains the EMF-based models of all the important components of our system model such as the events, the hardware micro-tasks, the shared memory as well as the I/O modules. This EMF-based IR is also connected to the EMF-based IR (CDFG) of the micro-task synthesis design-flow for the retrieval of necessary information such as the global variables and I/O port addresses.

5.6.3 Extraction of guard expression for micro-task activation

Using all these pieces of information present in the IR, we derive the guard expression for each micro-task activation present in the system.

In simple form, a micro-task T_N can only be activated when the following conditions are met:

- All the internal and external event signals present in T_N's event configuration (or their logical combinations) are evaluated to true.
- The event signals (or their logical combinations) present in the event configuration of a task T_M are false where T_M is such a micro-task that is sharing a write-access with T_N to a memory or I/O resource.

Using the above mentioned conditions, we derive the following guard expression for a micro-task T_N's activation:

$$C_{G_N} = E_{A_N} \& \forall_{T_M} not(E_{A_M}) \& \forall_{T_I} not(E_{A_I}) \tag{5.1}$$

where E_{A_N} is the event configuration for T_N activation, T_M is a micro-task sharing a write-access with T_N to a memory resource and T_I is a micro-task sharing a write-access with T_N to an I/O resource. The command signals, generated by combinational logic (see Figure 5.7), that are used to control the status registers are evaluated by the guard expression derived above. The status registers are in turn connected to the power-gating ports of the corresponding hardware micro-tasks to control their activation/deactivation.

5.6.4 Hardware generation

In the final step, using the guard expressions evaluated for different micro-tasks and shared memories, an RTL EMF-model of a datapath is generated following the generic template given in Figure 5.7. This model contains the combinational logic for micro-task/memory activation, a set of 1-bit status and/or event registers that store the

signals until used by the micro-tasks, and I/O pads to communicate with different components of the system (such as micro-tasks, shared resources and I/O peripherals generating the events).

This EMF-model of the SM datapath is then processed through the facilities for code generation provided by the framework (e.g. the JET editor) to generate a synthesizable VHDL description for the SM.

5.6.5 C-simulator generation for early system validation

To speed-up the system-level validation of a micro-task-based WSN node, we have used our design-flow to generate a C-simulator for the SM. This C-simulator can be integrated with the C-codes of the WSN application and can work for an early validation and debugging for the SM-synthesis design-flow.

Just to demonstrate the power and area consumption of a generic SM generated through our design-flow, a simple example is presented in next section.

5.7 Experimental results of the SM generation design-flow

The detailed experimental validation of our design-flow in presented in Chapter 6 with the help of a case-study. However, just to demonstrate how light-weight an SM can be in terms of power and area, we wrote the system description of the TFG shown in Figure 1.4 in our DSL. We then processed this system through our tool and generated a VHDL description of the hardware SM that controls the activation/deactivation of the four micro-tasks and the shared memory.

We then synthesized this VHDL description for 130 nm process technology and got its static and dynamic power consumption as well as the area overhead. According to the results, the SM hardware consumes 5.15 μW of dynamic power while operating at 16 MHz and a mere 296 nW of static power while using standard cell libraries at 1.2 V. The static power consumption can be further reduced to 80 nW if low-power cell libraries working at 0.3 V are used for 1-bit registers present in the hardware (see Figure 5.7). As far as the area overhead of the SM is concerned, it take only 754 μm^2 that corresponds to just 1% of the surface area consumed by an MSP430-core synthesized using the same process technology.

Next chapter contains further details about the experimental setup, the application benchmarks as well as our methodology for comparing the power/energy benefits of the proposed approach over a conventional MCU-based WSN-node.

Chapter 6

Experimental setup and results

This chapter presents the experimental setup and the results that we have achieved. It starts by describing the effects of using power-gating technique in our system and improvement achieved in wake-up response time. It then covers the dynamic and static power reductions obtained by our approach as compared to the currently available low-power MCUs in the light of a case study WSN application. Additionally, it provides the findings based on the design space exploration that we performed by varying the sizes and bitwidths of the micro-task datapath components and summarizes the optimal option. This chapter also provides the results for the power consumption of a hardware SM controlling the micro-tasks present in the case study and compares them with a soft-core MCU-based solution. The chapter finally concludes with the analysis of overall energy gain obtained while using power-gating over an entire period of a power-gated micro-task activation.

6.1 Power-gating and resultant switching delays

To check the applicability of power-gating in our proposed node architecture, we used a similar model of power-gated blocks as was used by Hu et al. [59]. However, as the authors did not provide any quantitative data for the switching delays specific to a CMOS technology, we had to re-run the experiments.

For this purpose, *Eldo* from Mentor Graphics was used to perform the transistor-level SPICE simulations using a 130 nm CMOS technology at a supply voltage of 1.2 V[1]. We used parallel NAND gates to model the timing behavior of a gated block (as shown in Figure 6.1). We observed a linear relation between the number of gates to be power-gated, n and the gating-transistor width, W. Similarly, a linear relation between n and output switching delays of the circuit was observed if the width of the gating-transistor was kept constant. Figure 6.2 shows this relation for a transistor width of 2.04 μm. It means that we are constrained on the number of gates to be driven by a single gating-transistor, if the switching delays are to be kept small.

[1] At that time, the design kit in 65nm was not available in the Lab.

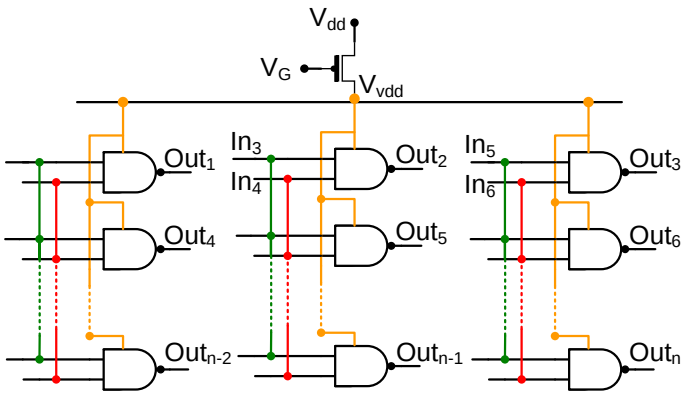


Figure 6.1: Parallel NAND gates model used to perform the SPICE transistor level simulations.

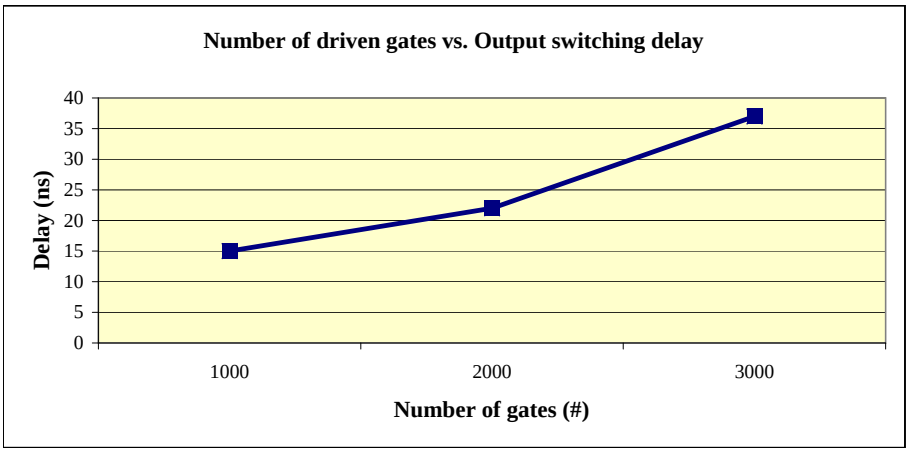


Figure 6.2: Linear relation between the number of gates being driven by a gating-transistor and the output switching delay (0 to 1).

On the other hand, if the total number of gates driven by a gating-transistor was kept constant, an inverse linear relation was found between the width of the gating-transistor and the output switching delay (as shown in Figure 6.3). This also shows that if the overall output switching delays are to be kept small, we have to either increase the width of the gating-transistor or limit the number of gates driven by a single gating-transistor.

We used a logic block of 3000 gates that is comparable in silicon area to the largest hardware micro-task present in our case study application and large enough for normal WSN control tasks (see Section 6.3). Figures 6.4 (a and b) show the turn-off and turn-on delays for the gate-outputs with a gating-transistor width of 2.04 μm. We chose this width as it permits the reasonable output switching delays with a moderate size of gating-transistor.

The early results show that we have a turn-on delay of 37.6 ns and turn-off delay

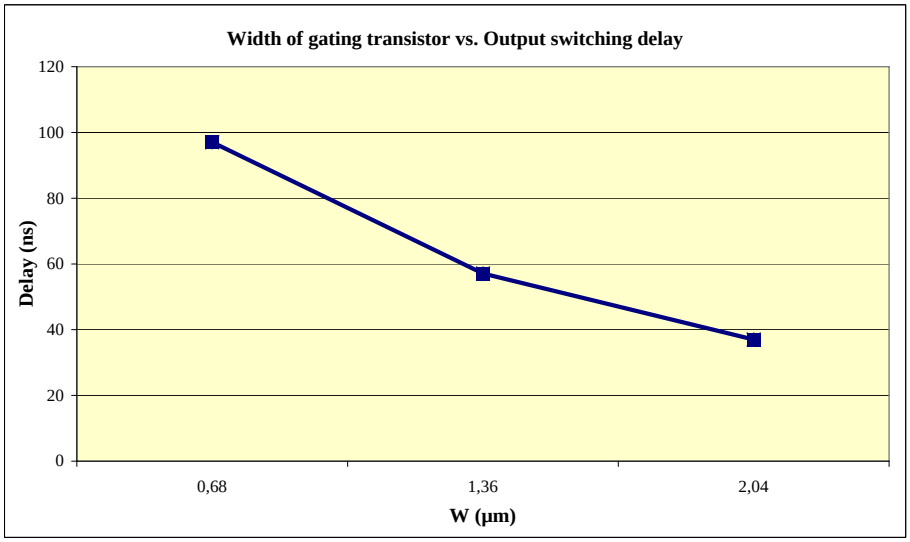


Figure 6.3: Inverse linear relation between the width of the gating-transistor and the output switching delay (0 to 1) for ($n = 3000$).

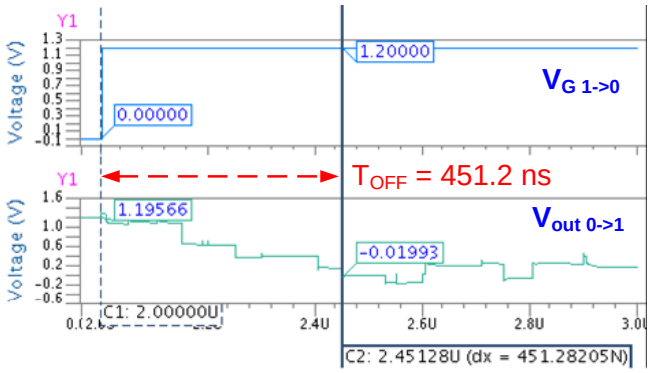
of 451 ns (between cut-off and active mode). This must be compared with MSP430's typical wake-up delay of 1 μs from the standby mode [129]. Further work is being done in our group on accurate modeling and estimation of wake-up delays and wake-up energy estimations of power-gated clustered [26].

6.2 An illustrative WSN application

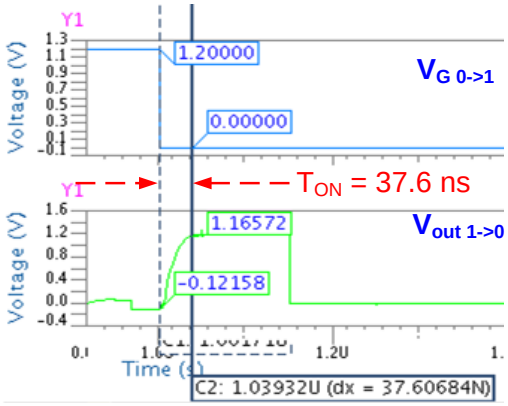
In order to perform power, area and energy comparison of our proposed micro-task-based WSN node with the traditional low-power MCU-based WSN node, we need a case-study WSN application whose application and control-oriented tasks can be run on the two platforms. For this purpose, we have to take a closer look at the applications targeted by WSN technology. Hence, this section starts with a short bibliographical study of the existing WSN application domains. We then move on to the established application benchmarks existing for WSN and finally extract a case-study application for the comparison.

6.2.1 Existing WSN applications

In the last decade, a wide range of application domains for WSN have been developed. Some of the application areas are environment, military, health, and security. WSN may include many different types of sensors such as seismic, low sampling rate magnetic, thermal, visual, infrared, acoustic and radar. These sensors are able to monitor a wide variety of ambient conditions such as temperature, humidity, lightning, pressure, and vehicular movements etc [35]. This section details the literature study of some of such



(a)



(b)

Figure 6.4: The output turn-on and turn-off delays for ($n = 3000$).

WSN applications.

Environmental-health monitoring is an important application of WSN. A lot of research work has been done on the different environmental aspects. Mainwaring et al. [89] developed a habitat monitoring system that monitors the habitats of birds, animals and insects. Similarly, forest fire detection and prevention [82], strength monitoring of the civil infrastructures [72], and detection of volcanic eruptions [142] are some other examples of environment-monitoring WSN systems.

WSN can also be used as an integral part of military Command, Control, Communication, Computing, Intelligence, Surveillance, Reconnaissance and Targeting (C4ISRT) systems [4]. The rapid deployment, self-organization and fault tolerance are some characteristics that make WSN a very promising sensing technique for military *C4ISRT* systems. Similarly, VigilNet is also a good example of an integrated wireless sensor node for military surveillance application [52]. VigilNet acquires and verifies information about enemy capabilities and positions of hostile targets.

In addition, the benefits of WSN have also been proved in other domains of human life such as health and home applications ([90], [54]).

6.2.2 WSN application benchmarks

It can be clearly seen that WSN applications consist of a heterogeneous nature as they are pretty different in their overall goals. However, the basic tasks performed in a WSN node are quite similar. These tasks are: sensing a certain phenomenon, gathering its relevant data and forwarding it to a base-station in a pre/post-processed state. Several attempts have been made to profile the workload of a generic WSN node. Two of the recent application benchmarks for WSN are SenseBench [97] and WiSeNBench [96]. Both of them covered majority of the general applications and algorithms that can be run on a typical WSN node.

To cover the OS-task aspect, we also used several of the OS-related control-tasks such as a next-node calculation function used in multi-hop geographical routing algorithm (similar to that was used in our group by PowWow [64]), and the drivers used to exchange data with the SPI-interface of RF transceiver such as CC2420. All of the above-mentioned application and OS tasks provide an adequate database of real-life WSN applications mostly used by WSN designers.

Using this data-base of application codes, we have generated a simple yet realistic example of a WSN application example that will serve us the purpose of illustrating power and energy gains of our approach. The next section discusses in details about different software tasks that are part of this case study and are used to generate the hardware micro-tasks through micro-task synthesis design-flow of *LoMiTa*.

6.2.3 The case study

This section highlights the important control tasks of our lamp-switching WSN application (discussed briefly in Section 5.1.3) during transmit as well as receive mode. The control-flow of the proposed node is based on RICER (Receiver Initiated CyclEd

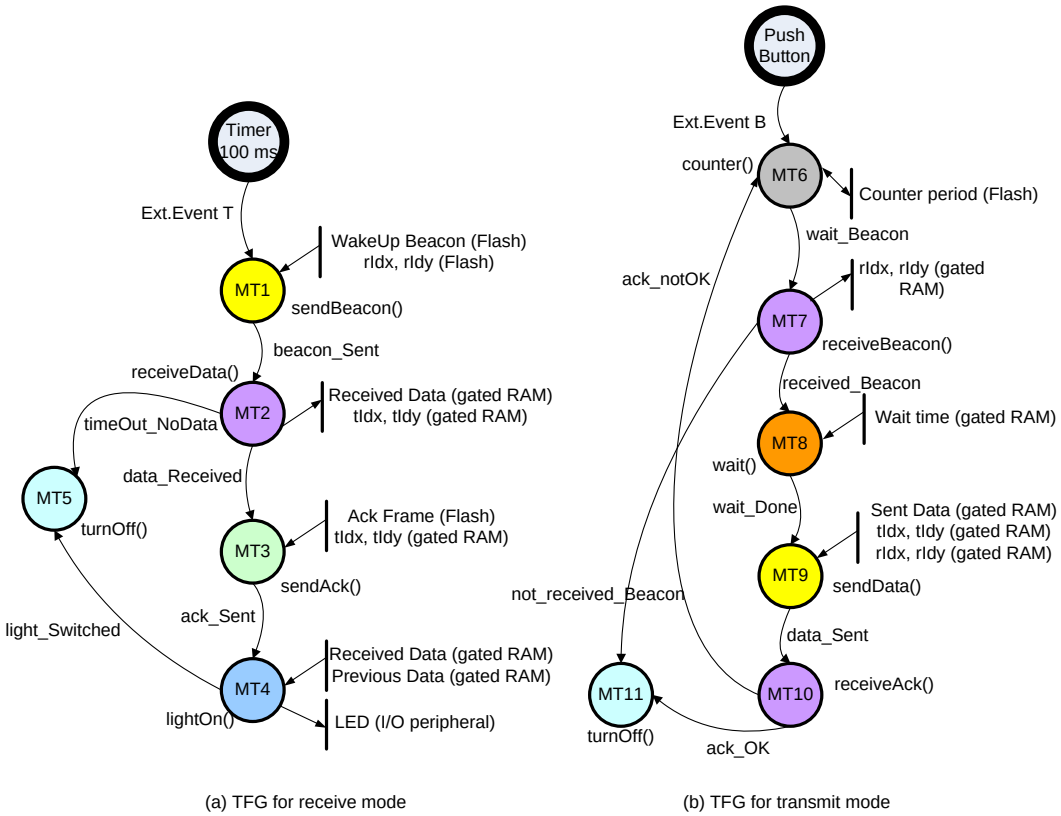


Figure 6.5: TFGs presenting the micro-tasks running during a lamp switching application.

Receiver) MAC protocol [84]. Briefly speaking, data transmission by a transmitter node is initiated upon reception of a wake-up beacon from the desired receiver node. Figures 6.5 (a and b) shows the TFGs of the proposed node in transmit and receive mode, respectively.

6.2.3.1 Tasks running in transmit mode

The basic control tasks running in our WSN node example in transmit mode are as follows:

- Wait for wake-up beacon: Upon reception of an external event (*Ext.Event B*) from push-button or an *ack_notOK* event, the transmitter node checks an internal counter if it has already achieved the maximum number of attempts to send a packet according to RICER protocol. We have written a C-function, called `counter()`, that actually checks that maximum number. If the counter is in valid state, a *wait_Beacon* event is generated to start the next task.

- Receiving the beacon frame: The next task in TFG is receiving the beacon, the

transmitter node waits for the wake-up beacon from a receiver. We have written a C-function, called `receiveBeacon()`, that actually starts a timer and reads the data packets received by its RF transceiver through SPI-link. In our example, we are using CC2420 radio chip from Texas Instrument [131] as RF transceiver. If the transmitter receives the required wake-up beacon before the timer expiry, it generates a *received_Beacon* event. Otherwise, a *not_received_Beacon* event is generated by the task.

- Waiting for channel clearance: Upon reception of *received_Beacon* event, the `wait()` function is activated. This task waits for a pre-defined wait period according to RICER protocol for the communication channel clearance and generates a *wait_Done* event.

- Sending data: The next task is data transmission that is described in `sendData()` function. This function writes data frame to the physical interface of the radio transceiver through SPI bus and generates a *data_Sent* event.

- Receiving acknowledgment: After sending data, the transmitter node waits for an acknowledgment frame from the receiver using `receiveAck()` function. If it receives the acknowledgment correctly, it generates an *ack_OK* event, otherwise an *ack_notOK* event is generated.

- Shutting down the transceiver: Upon reception of *ack_OK* or *not_received_Beacon* event, the transmitter node will shut down its RF transceiver to save energy. This is done through `turnOffRadio()` function, that sends appropriate signals to RF transceiver to shut it down.

6.2.3.2 Tasks running in receive mode

The control tasks running on a WSN receiver node of our case study are as follows:

- Sending a wake-up beacon: Our proposed WSN node periodically broadcasts a wake-up beacon to invite the neighbors to initialize a communication. This control task waits for an external event *Ext.Event T* for its activation. This external event is periodically generated by a hardware timer. The corresponding C-function for the task is described in `sendBeacon()` function that generates a *beacon_Sent* event.

- Receiving and analyzing data: After sending the beacon, the receiver waits for the data frame from any transmitter. The task is described in `receiveData()` function that starts a timer for possible time-out and receives and analyzes the data frame if it is destined for the receiver node or not. In case of valid data, it generates *data_Received* event whereas if the timer is expired and no valid data is received, a *timeOut_NoData* event is generated.

- Sending acknowledgment: Upon successful data reception, the receiver generates an acknowledgment for the transmitter node by calling `sendAck()` function that

sends an acknowledgment frame to its RF transceiver and generates a *ackSent* event.

- Switching the lamp: The next task, in receiver TFG, is switching the lamp that is accomplished by calling the `switchLamp()` function. This function analyzes the previous state of the lamp by reading its corresponding port and inverses it to switch the lamp state. Then it generates a *lamp_Switched* event.
- Shutting down the transceiver: Upon reception of *lamp_Switched* or *timeOut_NoData* event, the receiver node shuts down its RF transceiver to conserve energy until the next periodic wake-up event.

All the tasks present in the TFGs of this case-study were processed through hardware micro-task synthesis design-flow to generate hardware micro-tasks of two different bitwidths (8-bit and 16-bit). The resultant VHDL descriptions for the specialized hardware were synthesized through back-end commercially available synthesis tools. In next section, we discuss the synthesis results obtained during our experiments.

6.3 Dynamic power gains

The hardware micro-task (FSM + datapath) VHDL descriptions have been synthesized for both 130 nm and 65 nm CMOS process technologies using *Design Compiler* from Synopsys. We used these synthesis results to extract gate-level static and dynamic power estimations where an operating frequency of 16 MHz was assumed. These results were compared to the power dissipated by two different versions of MSP430 MCU:

- *tiMSP*, the Texas Instruments MSP430F21x2. We used the datasheet information for its power consumption (8.8 mW @16 MHz in active mode) which includes memory and peripherals,
- *openMSP*, an open-source MSP430-core without accounting for program memory, data memory and peripherals. We synthesized it and did the statistical power estimation for 130 nm technology and found it to be 0.96 mW @ 16 MHz.

We expect the actual power dissipation of the MSP430-core along with its program memory to lie somewhere in between the two results, and compare our results to both of them.

The results are given in Table 6.1 through Table 6.5. Table 6.1 shows the machine-instruction and cycle count, time taken, power and energy consumption for both *tiMSP* and *openMSP* (for software implementation). Table 6.2 and Table 6.3 show the power and energy benefits for 8-bit hardware micro-tasks over both the MSP430 implementations for 130 nm and 65 nm technology respectively. Similarly, Table 6.4 and Table 6.5 summarizes the similar results for 16-bit hardware micro-tasks. Before comparing these results, we explain the methodology that we used to extract information about the cycle-count and total execution time.

6.3.1 Extraction of cycle count

The MSP430 ISA consists of several complex instructions and addressing modes that have multi-cycle execution time. There are two techniques to profile an application running on an MCU: (i) simulation-based technique and (ii) statistical technique.

In simulation-based technique, we use the instruction-set simulator (ISS)) of the MCU under-test that contains a counter to accumulate the number of cycles used by each machine instruction. We then simulate the byte-code contained in the instruction memory to accumulate the cycle count taken by the input code. The approach gives us the exact number of clock-cycles being used by an application code on a certain MCU.

In statistical technique, we statistically analyze assembly-level machine-code of the application and estimate an approximate value of cycle count as the exact control-flow of the application (such as the loops and branches) is not explored.

We used the statistical approach to get an approximate cycle count of the application codes of the micro-tasks running on an MSP430-core. Since, the application codes of some micro-tasks such as `sendFrame()`, `swithLamp()` or `receiveFrame()` involve communications with the I/O-peripherals (e.g. lamp and RF-transceiver) through I/O-ports, it is difficult to simulate their behavior using an ISS for the MSP430. As a consequence, we statistically added the number of cycles taken by all the machine instructions by analyzing them one-by-one. To handle the control-flow of the application (such as *if*-blocks and *for*-loops), we accumulated the cycles taken by both the branches of an *if*-block and accumulated the clock-cycles taken by all the instructions present in a *for*-loop for only one iteration.

Similarly, the cycle count for the hardware micro-task is also performed using the statistical technique where the number of clock-cycles taken is equal to the number of FSM-states present the hardware micro-task.

Finally using this information about the approximate cycle count and a given common operating frequency, the total execution time for a micro-task running on the MSP430 as well as implemented in hardware as a hardware micro-task was estimated.

Going back to Table 6.2 through Table 6.5, it can be observed that, for different benchmark applications and OS tasks, our approach gains between one to two orders of magnitude in power and energy consumption. As it was discussed in Section 2.5.3, the other important parameter for a low-power MCU is its energy efficiency in terms of *Joules/instruction*. The energy efficiency of our approach is discussed in the next section.

6.3.2 Approximate energy efficiency

As mentioned by Hempstead et al. [53] that the energy efficiency measurement for the accelerator-based (hardware specialization) implementations are hard to evaluate in terms of *Joules/instruction*. Hence, they introduced the notion of *Joules/task*. Similarly, as the hardware micro-tasks generated through our design-flow are also not instruction-set processors, their exact energy-efficiency in terms of "Joules/instruction"

Task Name	MSP430						
	Instr. Count	Clk Cycles	time (μs)	Power (mW)		Energy (nJ)	
				tiMSP	openMSP	tiMSP	openMSP
crc8	30	81	5.1	8.8	0.96	44.9	4.9
crc16	27	77	4.8	8.8	0.96	42.2	4.6
tea-decipher	152	441	27.5	8.8	0.96	242	26.4
tea-encipher	149	433	27.0	8.8	0.96	237.6	26
fir	58	175	10.9	8.8	0.96	96	10.4
calcNeigh	110	324	20.2	8.8	0.96	177.7	19.4
sendFrame	132	506	31.6	8.8	0.96	278	30.3
receiveFrame	66	255	15.9	8.8	0.96	139.9	15.2

Table 6.1: Power/energy consumption of MSP430 for different application tasks (@ 16 MHz).

Task Name	8-bit Micro-task								
	No. States	time (μs)	Power (μW)	Energy (pJ)	P. Gain (x) P1/P2	E. Gain (x) E1/E2	Area (μm^2)	Eq. No. Nand Gates	E.Eff. (pJ/Inst.)
crc8	71	4.4	30.09	132.4	292/32	339/37	5831.7	730	4.4
crc16	103	6.4	46.92	300.3	187/20.4	140.5/15.3	8732.5	1092	11.1
tea-decipher	586	36.6	84.5	3090	104/11.4	78/8.55	19950	2494	20.3
tea-encipher	580	36.2	87.3	3160	101/11	75/8.2	20248	2531	21.2
fir	165	10.3	75.3	775.6	116/12.8	123.8/13.4	13323.7	1666	13.3
calcNeigh	269	16.8	74.3	1248.2	118/12.9	142.4/15.5	14239.4	1780	11.3
sendFrame	672	42	33.3	1400.3	264/28.8	198.5/21.7	10578	1323	10.6
receiveFrame	332	20.7	27.3	565	322/35	247.6/26.7	5075.3	635	8.5

Table 6.2: Power and energy gain of 8-bit micro-tasks over MSP430 (@ 16 MHz, 130 nm). Here, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

can not be measured.

However, since these micro-tasks are comparable to the MSP430 in terms of their overall execution time, we used the instruction count for MSP430 implementation from Table 6.1 and the actual energy consumptions of the hardware micro-task (for each application and control task) to calculate equivalent energy efficiency for them. The values are given in the last column of respective tables (Table 6.2 through Table 6.5). It can be observed that even if the operating voltage is neglected, our hardware micro-tasks are better (in terms of energy efficiency) than the WSN-specific subthreshold processors that are manually designed and optimized for ultra low-power WSN domain. In addition, if we scale the results using a common subthreshold voltage then we would do a lot better than these approaches. Table 6.6 reflects this fact where the actual and normalized energy efficiencies of various WSN-specific processors are compared with that of an average hardware micro-task.

Task Name	8-bit Micro-task							
	No. States	time (μs)	Power (μW)	Energy (pJ)	P. Gain (x) P1/P2	E. Gain (x) E1/E2	Area (μm^2)	E.Eff. (pJ/Inst.)
crc8	71	4.4	8.0	35.3	1095/32	1272/37.3	1762	1.2
crc16	103	6.4	12.4	79.2	710/20.6	532.8/15.5	2678	2.9
tea-decipher	586	36.6	22.6	827	389/11.32	292.6/8.6	6138	5.4
tea-encipher	580	36.2	23.3	845	377/10.99	281/8.27	6230	5.6
fir	165	10.3	20.4	209.7	432/12.54	458/13.3	4124	3.6
calcNeigh	269	16.8	20.1	337.8	437/12.73	526/15.4	4454	3.1
sendFrame	672	42	8.84	371.3	995/29	748/21.7	3434	2.8
receiveFrame	332	20.7	7.4	53.2	1189/34.6	913/26.8	1561	0.8

Table 6.3: Power and energy gain of 8-bit micro-tasks over MSP430 (@ 16 MHz, 65 nm). Here, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

Task Name	16-bit Micro-task								
	No. States	time (μs)	Power (μW)	Energy (pJ)	P. Gain (x) P1/P2	E. Gain (x) E1/E2	Area (μm^2)	Eq. No. Nand Gates	E.Eff. (pJ/Inst.)
crc8	71	4.4	55.3	242.6	159.6/17.4	185.1/20.2	10348	1294	8.1
crc16	73	4.56	55.0	251.0	159.8/17.4	168.1/18.3	10280	1285	9.3
tea-decipher	308	19.2	152.8	2940	57.6/6.2	82/9	27236	3405	19.3
tea-encipher	306	19.1	152.3	2910	57.8/6.3	81/8.93	27069	3384	19.5
fir	168	10.5	144.2	1514	61.02/6.7	63.4/6.9	23547	2944	26.1
calcNeigh	269	16.8	142.4	2392	61.8/6.7	74.3/8.1	24745	3094	21.7
sendFrame	672	42	58.1	2440	151.5/16.5	114/12.4	14863	1858	18.5
receiveFrame	332	20.7	50.0	1036	175.8/19.2	135/14.7	9485	1183	15.7

Table 6.4: Power and energy gain of 16-bit micro-tasks over MSP430 (@ 16 MHz, 130 nm). Here again, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

Task Name	16-bit Micro-task							
	No. States	time (μs)	Power (μW)	Energy (pJ)	P. Gain (x) P1/P2	E. Gain (x) E1/E2	Area (μm^2)	E.Eff. (pJ/Inst.)
crc8	71	4.4	14.71	64.72	598.2/17.4	693.7/20.3	3097	2.1
crc16	73	4.56	14.69	66.98	599/17.4	630/18.4	3102	2.5
tea-decipher	308	19.2	40.85	784.3	215.4/6.3	308.5/9.04	8380	5.1
tea-encipher	306	19.1	40.61	776.0	216.7/6.3	306.2/9	621	5.2
fir	168	10.5	39.03	409.8	225.5/6.56	234.3/6.8	7164	7.0
calcNeigh	269	16.8	38.58	648.1	228/6.4	274/8	7613	5.9
sendFrame	672	42	15.53	652.2	566.6/16.5	426/12.52	4771	4.9
receiveFrame	332	20.7	13.67	283.0	643.7/18.72	494/14.42	2858	4.3

Table 6.5: Power and energy gain of 16-bit micro-tasks over MSP430 (@ 16 MHz, 65 nm). Here again, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

Processor	**Operating Voltage (V)**	**Actual Energy (pJ/instruction)**	**Normalized Energy (pJ/instruction)** *(@ 0.50 V)*	**Process Technology**
SNAP/LE [31]	0.60	75	52	180
RISC-like core [79]	0.50	27	27	65
Charm [123]	1.03	96	23	130
BlueDot [119]	NA	26	NA	130
Hardware micro-task	1.20	12.4	2.2	130

Table 6.6: Actual and normalized energy-efficiencies for various ultra low-power WSN-specific processors.

Bit Width	Register File Depth	RAM Depth	ROM Depth	ALU Fns.	Power (μW)		Area (μm^2)	
					130 nm	65 nm	130 nm	65 nm
8	4	4	4	8	57	16.76	7635	2159
8	8	0	8	6	49.7	14.17	7038	2093
8	16	0	2	4	69	20.2	11163	3387
8	4	2	2	2	42	12.23	6160	1617
8	16	2	4	6	93	26.8	13098	4034
16	8	0	4	6	99	27.8	13184	4013
16	4	4	4	8	116	34.05	14423	4199
16	16	0	2	4	139.5	40.73	21590	6555
16	4	2	2	2	88	25.62	11715	3251

Table 6.7: Power consumption for datapaths having different design parameters (@ 16 MHz).

6.4 Design space exploration for datapath bitwidth

Our proposed software design-flow was used to generate hardware micro-tasks having both 8-bit and 16-bit datapaths to monitor the power savings as compared to commercial MCUs such as the MSP430.

We synthesized the customized datapaths of different hardware micro-tasks extracted from the above-mentioned benchmarks. The synthesis was again performed for both 130 nm and 65 nm CMOS technologies and the power and area estimations are given in Table 6.7.

6.4.1 8-bit vs. 16-bit micro-task

The application codes under study have different wordlength operations. For example, the application *crc16* mostly uses 16-bit wordlength operations while operations in *tea-decipher* and *tea-encipher* involve 32-bit wordlength data. The rest of the applications under-test use 8-bit wordlength operations.

As expected, for application codes, having wordlengths greater than 8-bit, an 8-bit micro-task has twice the number of FSM states than a 16-bit micro-task due to the bitwidth adaptation step of our design-flow. However, interestingly the FSM of a micro-task consumes much lesser power than the datapath and power consumption of

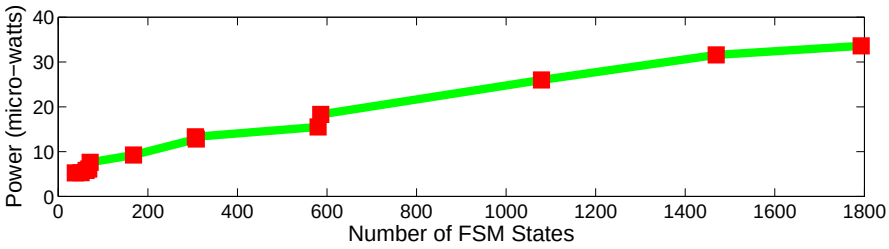


Figure 6.6: Power consumption vs. number of states of a micro-task FSM.

even very large FSMs increases in a sub-linear fashion with the number of states as shown in Figure 6.6.

As a result, an 8-bit micro-task consumes nearly half the power and silicon area than a 16-bit micro-task (Figures 6.7 (a and b)). As far as the energy consumption is concerned, for codes having wordlengths greater than 8-bit, total energy consumption of an 8-bit and 16-bit micro-task is nearly the same. On the other hand, for application codes having 8-bit wordlength, an 8-bit micro-task consumes half of that of a 16-bit micro-task, (as shown in Figure 6.7 (c)).

Hence, as the datapath power dominates the FSM power in our case study, an 8-bit micro-task is a better solution. Nevertheless, for cases where FSMs could be comparatively much larger and consume more power than the datapath, micro-tasks having larger bitwidth would become more suitable.

6.5 Power estimation of hardware system monitor

To compare the power consumptions and potential area overhead of a hardware system monitor (SM), we wrote a system-level description of the TFGs present in our case-study application using our DSL. Then we used our design-flow for the SM synthesis to process this system-level description and generated an RTL description of the SM controlling the hardware micro-tasks, I/Os and memory modules present in our proposed WSN-node (similar to the one shown in Figure 5.4).

6.5.1 Dynamic power consumption

The VHDL description of the hardware SM was synthesized for 130 nm process technology. The results show that it consumes only 12 μW at an operating frequency of 16 MHz and since the average active period of a WSN node is quite low (less than 1% of the overall duty cycle), this power consumption is negligible as compared to an OS-based scheduler running on an MSP430 that consumes between 8.8 mW (*tiMSP*) to 1 mW (*openMSP*).

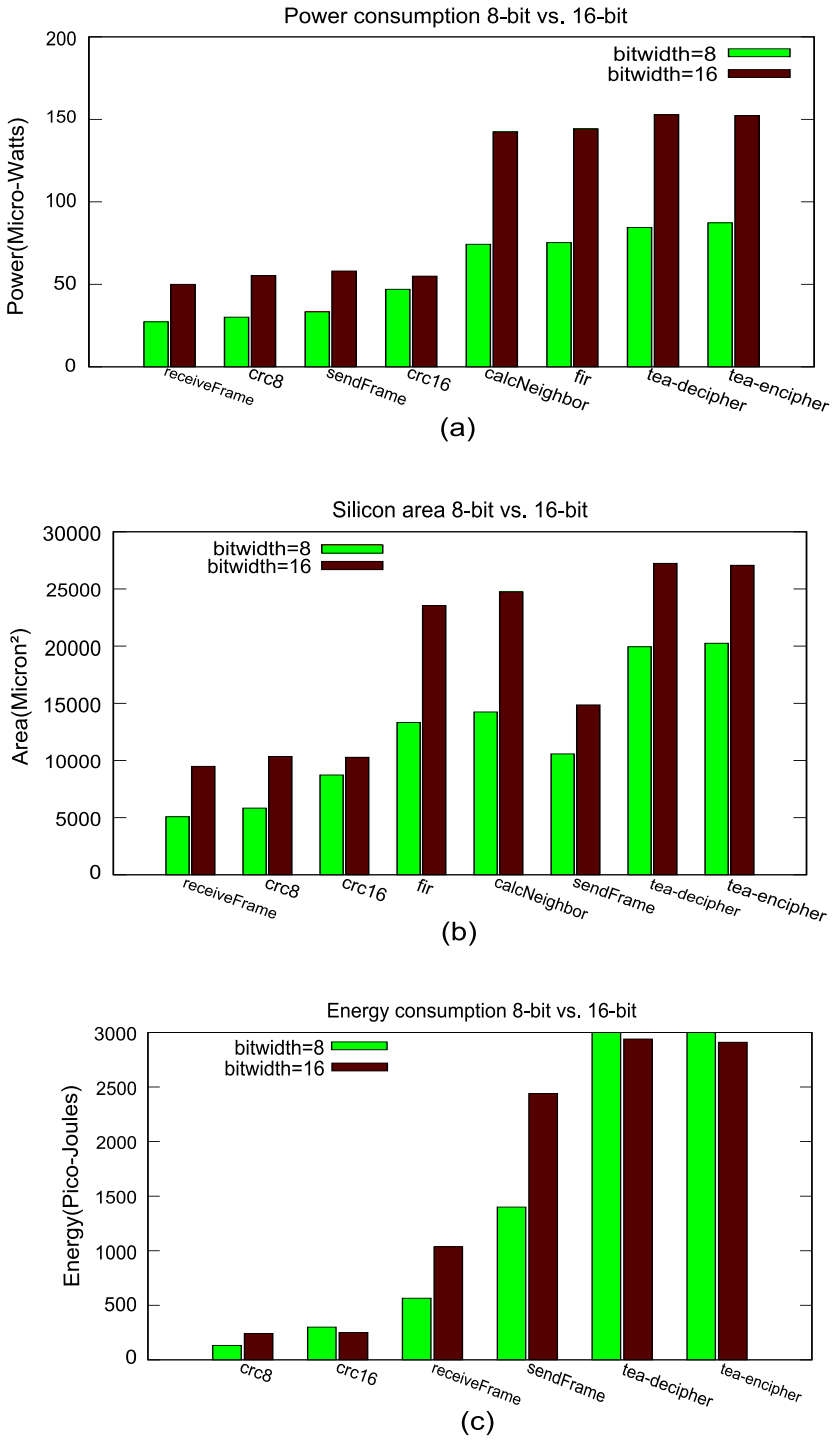


Figure 6.7: Comparison of power, area and energy consumption for 8-bit and 16-bit micro-tasks.

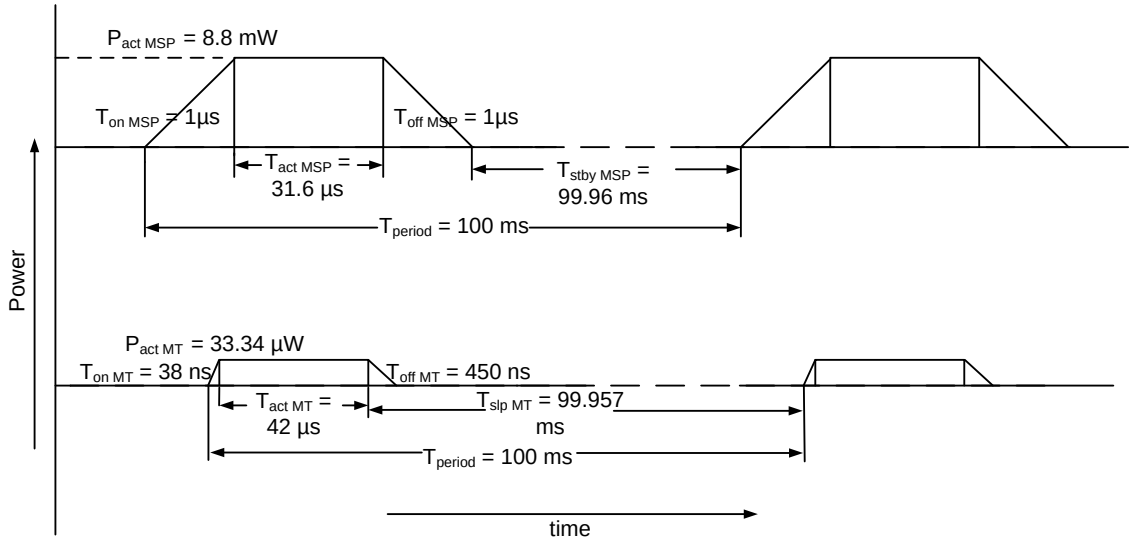


Figure 6.8: Time distribution of *sendFrame* task duty cycle.

6.5.2 Static power and area overhead

As far as the static power consumption of the synthesized SM is concerned, it consumes approximately 600 nW of static power when implemented with standard cell library. It is worth-noticing that the major part of this static power is consumed due to the various 1-bit flip-flops present in the SM. For the sake of exploration, we experimented with different available cell libraries (e.g. general purpose, low-power etc.) and different levels of V_{DD}. We found out that even using the standard cell libraries and just by lowering the V_{DD} to 0.3 V (the minimum voltage for data retention), the static power consumption of the SM can be reduced to approximately 150 nW whereas the static power consumption of the MSP430 is around 1.54 μW (from the datasheets).

Similarly, if we look at the area overhead of the hardware SM used in our case-study example, it is only 1710 μm^2 that corresponds to only 2% of area consumed by an openMSP-core synthesized for the same process technology.

Looking at these power and area results, obtained for the hardware SM, it is quite evident that for an area overhead of only 2%, we are gaining approximately 10x in terms of static power consumption whereas the dynamic power consumption of the hardware SM is negligible as compared to an equivalent OS-based software implementation. However, we must take into account the static power consumption of the power-gated hardware micro-tasks present in the system. The effects of static power consumption of a hardware micro-task on the overall energy gain during a micro-task activation is discussed in the next section.

6.6 Effects of low duty-cycle and overall energy gain

As mentioned earlier, a WSN-node has a very low active duty-cycle. If we look at Tables 6.2 through Table 6.5, they show the energy gains of the proposed technique during the active period only, whereas the overall energy gain of a micro-task-based WSN node can be obtained if complete time period of a task activation and the static energy dissipation during stand-by mode is considered.

Figure 6.8 shows the simplified version of time distribution for *sendFrame* task activation having a wake-up time period of 100 ms which means that our transmitting node is sending a packet to the RF transceiver through SPI-bus every 100 ms. Such a task can be considered as highly reactive w.r.t. WSN application domain. It can be can clearly observed that the node has a pretty low duty cycle (even less than 1% of the whole time period). The switching delays of the MSP430 are taken from its datasheet whereas those of the micro-task block from the Section 6.1.

The overall energy gain of our approach over the MSP430 can be presented by the following expression:

$$Gain_{tot} = \frac{E_{act_{msp}} + (P_{stby_{msp}} \times T_{stby_{msp}}) + (\frac{1}{2} P_{act_{msp}} \times (T_{on_{msp}} + T_{off_{msp}}))}{E_{act_{mt}} + (P_{slp_{mt}} \times T_{slp_{mt}}) + (\frac{1}{2} P_{act_{mt}} \times (T_{on_{mt}} + T_{off_{mt}}))} \tag{6.1}$$

where

- $E_{act_{msp}}$ is the dynamic energy of the MSP430 (given in Table 6.1),
- $P_{stby_{msp}}$ is the static power consumption of the MSP430,
- $T_{stby_{msp}}$ is the time spent in standby mode by the MSP430,
- $P_{act_{msp}}$ is the dynamic power of the MSP430 (given in Table 6.1),
- $T_{on_{msp}}$ is the turn-on delay of the MSP430 (given in [129]),
- $T_{off_{msp}}$ is the turn-off delay of the MSP430 (given in [129]),
- $E_{act_{mt}}$ is the dynamic energy of the micro-task (given in Table 6.2)
- $P_{slp_{mt}}$ is the static power consumption of the micro-task,
- $T_{slp_{mt}}$ is the time spent in sleep mode by the micro-task,
- $P_{act_{mt}}$ is the dynamic power of the micro-task (given in Table 6.2),
- $T_{on_{mt}}$ is the turn-on delay of the micro-task (c.f. Section 6.1),
- $T_{off_{mt}}$ is the turn-off delay of the micro-task (c.f. Section 6.1).

The MSP430F21x2 consumes approximately 1.54 μW in stand-by mode having a 512 Bytes of RAM. The static power of a micro-task in power-gated mode depends on the size of its RAM. We considered the same static power consumption (as that of the MSP430) for a micro-task and just scaled it down since a micro-task needs much smaller global memory (6 Bytes on average, see Table 6.7). Hence, average static power consumption of a micro-task would be around 18 nW.

Considering this static power, a time period of 100 ms and using the expression derived above, we calculated the overall energy gain for *sendFrame* micro-task over a complete period of task-activation. As a result, we gain approximately 138 x over the MSP430.

To conclude, as our system uses much lower power in sleep mode (thanks to power gating) and relatively shorter output switching delays than a low-power MCU (e.g. the MSP430), one to two orders of savings in energy are obtained when a complete period of a WSN task activation is considered. With these results, we close our discussion on experimental setup and results obtained for static and dynamic power consumptions of hardware the micro-tasks and the SM generated through our design-flows.

In next chapter, we conclude the work presented in this thesis and draw some future research directions.

Chapter 7

Conclusion and future perspectives

WSN is a fast evolving technology with a number of potential applications in various domains of human-life. Structural-health and environmental monitoring, medicine, military surveillance, smart environments and robotic explorations can be some examples of these domains. Recent advancements in mechanics, wireless communication, and digital electronics have enabled us to develop low-cost, low-power and multi-functional sensor nodes that are small in size and communicate efficiently over short distances. Systems of 1000s or even 10,000s of such nodes are anticipated.

WSN nodes are low-power embedded devices consisting of processing subsystem (an MCU connected to a RAM and/or flash memory), wireless communication subsystem (RF transceiver), power supply subsystem (power source and DC-DC converter) and sensory subsystem (sensor/actuator). Since WSN nodes must be small in size due to limited production cost, it is not possible to provide them with large power sources. In most cases they must rely on non-replenishing (e.g. batteries) or self-sufficient (e.g. solar cells) sources of energy. Hence, ultra low-power becomes the most critical design metric for a WSN node. It is also supported by the fact that WSN nodes may have to work unattended for long durations due to a large number of deployed nodes or a difficult access to them after deployment.

If we analyze the power consumption profile of a generic WSN node, among all of its subsystems, communication and computation subsystems consume bulk of the node's available power-budget. As a result, in this work, we targeted the power optimization of the computational subsystem of a WSN node.

As far as their design is concerned, WSN node computational and control subsystems have until now been based on low-power MCUs such as the MSP430 from Texas Instruments, the ARM Cortex-M0 by ARM and the ATmega128L from Atmel Corporation. These programmable processors provide a reasonable processing power with low power consumption at a very affordable cost. Most of such MCU-packages also offer a limited amount of RAM (from a few hundred Bytes to a few kilo-Bytes) and non-volatile flash memory.

However, these processors are designed for low-power operation across a wide range of embedded system application settings. As a consequence, they are not necessarily well-suited to WSN node design as they are based on a general purpose, monolithic compute engine. On the software end, WSN nodes generally rely on a light-weight OS layer to provide concurrency management for both external event handling and/or application task management. Eventually, power dissipation of current low-power MCUs still remains orders of magnitude too high for many potential applications of WSN.

We believe that the hardware specialization is an interesting way to further improve energy efficiency: instead of running the application/OS tasks on a programmable processor, we propose to generate an application specific micro-architecture, tailored to each task of the application at hand. We proposed such an approach where a WSN node computation subsystem is made of several hardware *micro-tasks* that are activated on an event-driven basis, each of them being dedicated to a specific task of the system (such as event-sensing, low-power MAC, routing, and data processing etc.). By combining hardware specialization with power reduction techniques such as *power-gating*, we drastically reduced both dynamic (thanks to specialization) and static (thanks to power-gating) power consumption.

In addition, our proposed micro-task-based WSN node contains a hardware scheduler, called *system monitor (SM)* to perform the task/power-management.

The key contributions of this research work can be described as under:

- We demonstrated using transistor-level SPICE simulations the potential benefit of power-gated specialized hardware and thus proposed the concept of "power-gated hardware micro-tasks".
- We thus showed that power-gating is applicable to our micro-task-based WSN node architecture and it happens to have very short switching-time delays, in the orders of a few hundred of nano seconds for the larger micro-tasks. This improves the wake-up response time by at least 50% when compared to low-power MCUs such as the MSP430.
- We provided an integrated design-flow for the synthesis of micro-task-based WSN node architectures. In this flow, the behavior of each micro-task is specified in C and is mapped to an RTL description of an application specific micro-architecture using a hybrid of retargetable ASIP-synthesis and HLS design methodologies.
- We also provided a DSL that can be used to specify the system-level description of WSN node (following an event-driven TFG). The second part of our design-flow, using this system-level description, generates hardware description for the SM that is used to control the activation and deactivation of the power-gated micro-tasks present on the WSN node.
- With the help of a simple yet realistic case-study of a WSN application, we showed that our approach provides power savings of one to two orders of magnitude in dynamic power when compared to the power dissipation of currently available MCU-based solutions.

- We also used our design-flow to perform design space exploration by exploring the trade-off in power/area that can be obtained by modifying the bitwidth of the generated micro-tasks, and compared the obtained results to those achieved by using an off-the-shelf low-power MCU such as the MSP430.

- Using the SM generation design-flow, we demonstrated that the hardware SM generated to control the micro-tasks and shared memories of our case-study application provides 10x reduction in static power and a negligible amount of dynamic power when compared to MCU-based solution. Moreover, the area overhead of a hardware scheduler is only 2% of the surface area of an MSP430-like core.

After enumerating the key contributions and results obtained during the course of this work, we take this opportunity to discuss some of the on-going advancements and future perspectives of current research work.

7.1 Work in progress

At the moment we are working on two different aspects of this research work. First one is the network-level validation of a micro-task-based WSN node. In our opinion, the generation of a C-simulator for the micro-tasks, that can be integrated in the existing C-simulators for WSN, is among the fastest ways to perform validation and debugging. There exist C-simulators for wireless sensor nodes and networks, such as WSim [62] and WSNet [63] that can be used for this validation. The basic approach is shown in Figure 7.1 where the C-simulators for different low-power MCUs and RF-transceivers are already developed in WSim. We are currently working on adding WSim-compatible C-simulator generation step to our design-flow for micro-task generation.

The second issue is the lack of programmability. Our approach has an obvious drawback: it assumes that the micro-tasks are hard-wired into silicon as ASIC blocks. This means that the behavior of each micro-task is fixed, making post-production upgrade or bug fixing very costly. This may look like a show stopper, as flexibility is often of a great concern for WSN system designers. However, when looking more carefully to actual design practices, we can observe that the need of flexibility and reprogrammability is essentially geared toward the user application layer, which happens to represent only a small fraction of the WSN node processing workload, this latter one being almost entirely dedicated to the communication stack.

Besides, in practice, designing a new WSN application usually means adapting a proved existing WSN software framework to a new user application. In other words, the communication stack software is generally reused "as is" and routing algorithms, MAC protocols, device driver layers remain the same (even if their behavior is parameterized). We therefore propose to combine the best of both worlds, that is:

- use a very small silicon footprint instruction-set processor (8-bit datapath, minimalistic RISC instruction set) with a power-gating feature to implement the application layer user software,

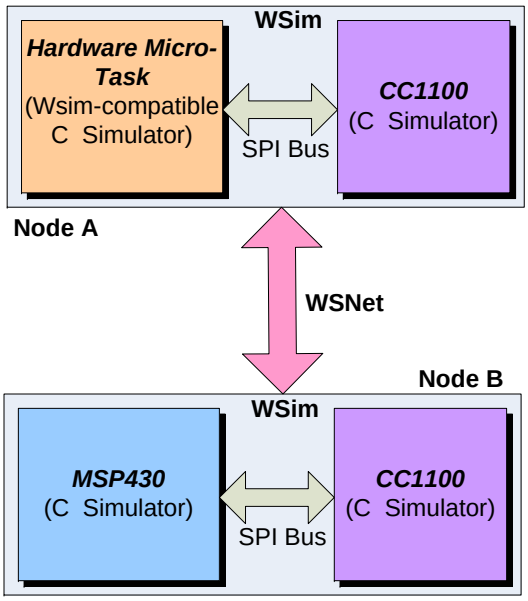


Figure 7.1: Network level validation of micro-task-based WSN node using WSim and WSNet.

- use a distributed system of micro-tasks to handle the OS-level services of the WSN node (mostly the communication stack).

Such an approach would preserve most of the power/energy savings provided by specialization, while preserving programmability at the user application level. Figure 7.2 shows the block diagram of a computation/control subsystem based on the above-proposed approach while Figure 7.3 shows the modified version of WSN node based on our approach after considering the issue of programmability.

We are currently working on the integration of our micro-task-based SoC to an MCU-based platform. A circuit in 65 nm process technology is under design and will be send to foundry around mid-November. It includes an *openMSP*-core and a basic micro-task for data-frame generation. We are also developing an interface between the MSP430 and our micro-task-based SoC. Beside this on-going work, there are some future perspectives as well that are discussed in the next section.

7.2 Future perspectives

Our approach generates a micro-task-based computation and control subsystem that is comparable in silicon area to an MCU-based approach with a relatively lower static power consumption (thanks to power gating).

However, it can be interesting to explore some ideas to further reduce the silicon area of a micro-task-based WSN node. Some of these ideas are discussed in the following paragraphs.

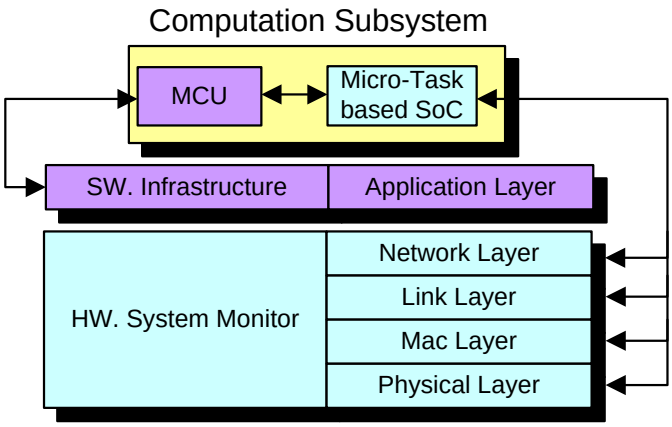


Figure 7.2: Proposed solution to tackle the issue of loss of reprogrammability.

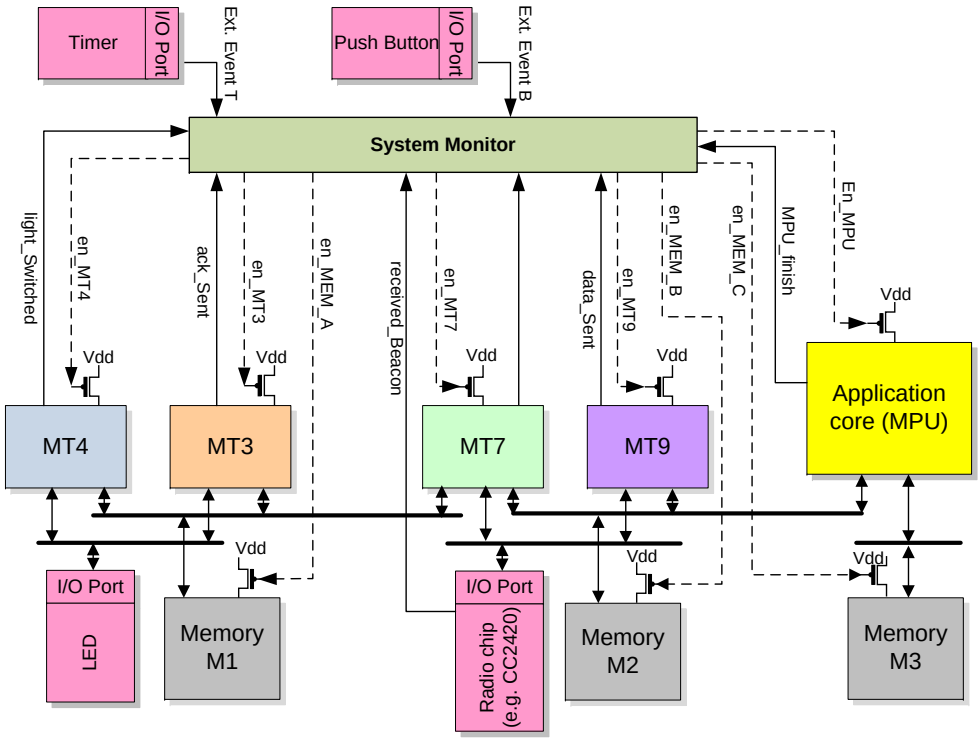


Figure 7.3: System-level view of a micro-task based WSN node architecture

Reconfigurable SoC

It will be interesting to study the portability of our approach on control-oriented reconfigurable structures, which would provide support for small grain power-gating techniques. Small grain partial power gating does not exist till now in commercial FPGAs but some work has been done in academics on this aspect [118].

Datapath merging

Datapath merging [94] has been used in reconfigurable architecture design to save the overall surface area. It could be interesting to study the feasibility of developing a hardware-OS based MSP430. In such an approach, the designed MSP430-core can work in two modes: (i) general-purpose mode and (ii) specialized mode. In general-purpose mode, the MSP430-core would perform the application-layer tasks of the communication stack. For lower layers of the communication stack, the MSP430-core would be used in specialized mode implementing a hardware-OS where the MSP430-datapath components such as the register file, and the ALU-operators could be used by the hardware micro-task FSMs generated through our design-flow to perform the control-oriented tasks such as event sensing, routing, packet processing and forwarding, etc. In this hardware-OS based MSP430, a fine-grain datapath merging could be applied to merge the datapath components present in hardware micro-tasks (generated through our design-flow) and the MSP430-datapath.

Since the hardware micro-tasks are power-gated, there will not be any extra cost for adding the multiplexers that are needed at the input of conventional datapath components being merged.

With some of the possible future perspectives discussed above, we conclude this manuscript with the publications extracted from this research work.

Personal publications

International conferences

- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "System Level Synthesis for Ultra Low-Power Wireless Sensor Nodes", EuroMicro DSD'10, Lille France, September 2010.
- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "A Complete Design-Flow for the Generation of Ultra Low-Power WSN Node Architectures Based on Micro-Tasking", ACM/IEEE DAC, Anaheim CA USA, June 2010.
- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "A Novel Approach for Ultra Low-Power WSN Node Generation", IET ISSC, Cork Ireland, June 2010.
- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "Toward Ultra Low-Power Hardware Specialization of a Wireless Sensor Network Node", IEEE INMIC, Islamabad Pakistan, December 2009 (Best Student Paper Award).
- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "Ultra Low-Power FSM for Control Oriented Applications", IEEE ISCAS, Taipei Taiwan, May 2009.

National workshops

- Muhammad Adeel Pasha, Steven Derrien, Olivier Sentieys, "Ultra Low-Power FSM for Sensor Networks", GDR SOC-SIP, Paris France, June 2009.

List of acronyms and abbreviations

ADC	**A**nalog to **D**igital **C**onverter
ADL	**A**rchitecture **D**escription **L**anguage
ALAP	**A**s **L**ate **A**s **P**ossible
ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASAP	**A**s **S**oon **A**s **P**ossible
ASIP	**A**pplication **S**pecific **I**nstruction-set **P**rocessor
BURS	**B**ottom **U**p **R**e-writing **S**ystem
BURG	**B**ottom **U**p **R**e-writing system **G**enerator
CAN	**C**ontroller **A**ea **N**etwork
C4ISRT	**C**ommand, **C**ontrol, **C**ommunication, **C**omputing, **I**ntelligence, **S**urveillance, **R**econnaissance and **T**argeting
CDFG	**C**ontrol **D**ata **F**low **G**raph
CGS	**C**oarse **G**rain **S**cheduling
CMOS	**C**omplementary **M**etal-**O**xide **S**emiconductor
COTS	**C**ommercially **O**ff **T**he **S**helf
CPU	**C**entral **P**rocessing **U**nit
CVS	**C**lustered **V**oltage **S**caling
DAG	**D**irected **A**cyclic **G**raph
DC	**D**irect **C**urrent
DFG	**D**ata **F**low **G**raph
DPU	**D**ata **P**ath **U**nit
DSL	**D**omain **S**pecific **L**anguage
EDF	**E**arliest **D**eadline **F**irst
EEPROM	**E**lectrically **E**rasable **P**rogrammable **ROM**
EMF	**E**clipse **M**odeling **F**ramework

FDLS **F**orce **D**irected **L**ist **S**cheduling
FDS **F**orce **D**irected **S**cheduling
FFT **F**ast **F**ourier **T**ransform
FGS **F**ine **G**rain **S**cheduling
FPGA **F**ield **P**rogrammable **G**ate **A**rrays
FPU **F**loating **P**oint **U**nit
FSM **F**inite **S**tate **M**achine
FSMD **FSM** with **D**atapath

GA **G**enetic **A**lgorithm
GeCoS **Ge**neric **Co**mpiler **S**uit

HAL **H**ybrid **AL**location
HLL **H**igh **L**evel **L**anguage
HLS **H**igh **L**evel **S**ynthesis

IAPO **I**nterconnect **A**ware **P**ower **O**ptimized
I^{2}C **I**nter **I**ntegrated **C**ircuit
ILP **I**nstruction **L**evel **P**arallelism
I/O **I**nput/**O**uput
IR **I**ntermediate **R**epresentation
ISS **I**nstruction **S**et **S**imulator
ISA **I**nstruction **S**et **A**rchitecture
ISE **I**nstruction **S**et **E**xtension

JET **J**ava **E**mitter **T**emplate

LEA **L**eft **E**dge **A**lgorithm
LoMiTa ultra **Lo**w-power **Mi**cro-**Ta**sking
LoS **L**ine **o**f **S**ight

MAC **M**edium **A**ccess **C**ontrol
MCU **M**icro-**C**ontroller **U**nit
MDE **M**odel **D**riven **E**ngineering
MEMS **M**icro **E**lectro **M**echanical **S**ystem
MILP **M**ixed **I**nteger **L**inear **P**rogramming
MIMO **M**ultiple **I**nput **M**ultiple **O**utput
MIPS **M**illion **I**nstructions **P**er **S**econd
MOS **M**antis **O**perating **S**ystem
MOSFET **M**etal-**O**xide **S**emiconductor **F**ield **E**ffect **T**ransistor
MTCMOS **M**ulti-**T**hreshold **CMOS**

NRE **N**o **I**ntruction **S**et **C**omputer
NRE **N**on-**R**ecurring **E**ngineering

OS	**O**perating **S**ystem
QoS	**Q**uality **o**f **S**ervice
RAM	**R**andom **A**ccess **M**emory
RF	**R**adio **F**requency
RICER	**R**eceiver **I**nitiated **C**ycl**E**d **R**eceiver
RISC	**R**educed **I**nstruction **S**et **C**omputer
ROM	**R**ead **O**nly **M**emory
RT	**R**egister **T**ransfer
RTL	**RT**-**L**evel
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SM	**S**ystem **M**onitor
SoC	**S**ystem **o**n **C**hip
SPI	**S**erial **P**eripheral **I**nterface
TFG	**T**ask **F**low **G**raph
UGH	**U**ser **G**uided **H**LS
ULIW	**U**ltra **L**arge **I**nstruction **W**ord
VHDL	**VHSIC** **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**cale **I**ntegrated **C**ircuit
VLIW	**V**ery **L**arge **I**nstruction **W**ord
VLSI	**V**ery **L**arge **S**cale **I**ntegrated
WSN	**W**ireless **S**ensor **N**etwork

Bibliography

[1] Actel Corporation. 2008. IGLOO Handbook. Tech. rep., Actel corporation.

[2] Aeroflex Gaisler. 2010. Leon4 Processor . Product.

[3] Aho, A. V., Ganapathi, M., and Tjiang, S. W. K. 1989. Code Generation using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages Systems 11,* 4, 491–516.

[4] Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. March 2002. Wireless Sensor Networks: A Survey. *Computer Networks 38,* 4.

[5] Alippi, C. and Galperti, C. 2008. An Adaptive System for Optimal Solar Energy Harvesting in Wireless Sensor Network Nodes. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications 55,* 6 (jul.), 1742–1750.

[6] Altera. 2010. Nios II Processor: The World's Most Versatile Embedded Processor.

[7] Anis, M., Mahmoud, M., Elmasry, M., and Areibi, S. 2002. Dynamic and Leakage Power Reduction in MTCMOS Circuits using an Automated Efficient Gate Clustering Technique. In *DAC'02: Proceedings of the 39th annual Design Automation Conference*. ACM, New York, NY, USA, 480–485.

[8] Atmel Corporation. 2007. ATmega 103L 8-bit AVR Low-Power Microcontroller. Tech. Report.

[9] Atmel Corporation. 2009. ATmega 128L 8-bit AVR Low-Power MCU. Tech. Report.

[10] Augé, Ivan and Pétrot, Frédéric. 2008. User Guided High Level Synthesis. In *High-Level Synthesis*, Coussy, Philippe and Morawiec, Adam, Ed. Springer Netherlands, Chapter 10, 171–196.

[11] B. Gorjiara and D. Gajski. 2008. Automatic Architecture Refinement Techniques for Customizing Processing Elements. In *DAC'08: Proceedings of the 45th annual ACM/IEEE Design Automation Conference*. ACM, Anaheim, USA, 379–384.

[12] Babighian, P., Benini, L., Macii, A., and Macii, E. 2004. Post-Layout Leakage Power Minimization Based on Distributed Sleep Transistor Insertion. In

ISLPED'04: Proceedings of the International Symposium on Low Power Electronics and Design. ACM, Newport Beach, California, USA, 138–143.

[13] Beutel, J., Kasten, O., and Ringwald, M. 2003. Poster Abstract: BTnodes – A Distributed Platform for Sensor Nodes. In *SenSys'03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 292–293.

[14] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. 2005. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mob. Netw. Appl. 10,* 4.

[15] C. A. Mandal, P. P. C. and Ghose, S. 1995. Complexity of Scheduling in High Level Synthesis. *VLSI Design 7,* 4, 337–346.

[16] Cadence. 2010. C-to-Silicon Compiler. Product.

[17] Cerpa, A., Elson, J., Hamilton, M., Zhao, J., Estrin, D., and Girod, L. 2001. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *SIGCOMM-LA'01: Workshop on Data communication in Latin America and the Caribbean*. ACM, New York, NY, USA, 20–41.

[18] Chaitin, G. 2004. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Not. 39,* 4, 66–74.

[19] Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. 1995. Low Power CMOS Digital Design. *IEEE Journal of Solid State Circuits 27*, 473–484.

[20] Chang, J. M. and Pedram, M. 1997. Energy Minimization Using Multiple Supply Voltages. *IEEE Transactions on VLSI Systems 5*, 436–443.

[21] Chavet, C., Andriamisaina, C., Coussy, P., Casseau, E., Juin, E., Urard, P., and Martin, E. 2007. A Design Flow Dedicated to Multi-mode Architectures for DSP Applications. In *ICCAD'07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, Piscataway, NJ, USA, 604–611.

[22] Chiou, D.-S., Juan, D.-C., Chen, Y.-T., and Chang, S.-C. 2007. Fine-Grained Sleep Transistor Sizing Algorithm for Leakage Power Minimization. In *DAC'07: Proceedings of the 44th annual ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, 81–86.

[23] Choi, E., Shin, C., Kim, T., and Shin, Y. 2008. Power-Gating-Aware High-Level Synthesis. In *ISLPED'08: Proceeding of the 13th International Symposium on Low power Electronics and Design*. ACM, New York, NY, USA, 39–44.

[24] Coussy, P., Gajski, D. D., Meredith, M., and Takach, A. 2009. An Introduction to High-Level Synthesis. *IEEE Design and Test of Computers 26,* 4, 8–17.

[25] Crossbow Technology. MICA2 Motes.

[26] D., V. T., Sentieys, O., and Derrien, S. 2011. Wakeup Time and Wakeup Energy Estimation in Power-Gated Logic Clusters. In *VLSI'2011: Proceedings of the 24th International Conference on VLSI Design*. Chennai, India.

[27] Deming Chen and Cong, J. and Yiping Fan. 2003. Low-Power High-Level Synthesis for FPGA Architectures. In *ISLPED'03: Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. 134 – 139.

[28] Devadas, S. and Malik, S. 1995. A Survey of Optimization Techniques Targeting Low Power VLSI Circuits. In *DAC'95: Proceedings of the 32nd ACM/IEEE conference on Design automation*. ACM, New York, NY, USA, 242–247.

[29] Dunkels, A., Gronvall, B., and Voigt, T. 2004. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN'04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. 455–462.

[30] Dutta, P., Grimmer, M., Arora, A., Bibyk, S., and Culler, D. 2005. Design of a Wireless Sensor Network Platform for Detecting Rare, Random, and Ephemeral Events. In *IPSN'05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. NJ, USA, 70.

[31] Ekanayake, V., Kelly, IV, C., and Manohar, R. 2004. An Ultra Low-Power Processor for Sensor Networks. *SIGOPS Oper. Syst. Rev. 38,* 5, 27–36.

[32] El-Hoiydi, A. and Decotignie, J.-D. 2004. WiseMAC: An Ultra Low Power MAC Protocol for the Downlink of Infrastructure Wireless Sensor Networks. In *ISCC'04: Proceedings of the 9th International Symposium on Computers and Communications*. 244–251.

[33] EM Microelectronic. 2005. EM6812, Ultra Low Power 8-bit FLASH Mircro-Controller. Tech. rep.

[34] Enz, C. C., El-Hoiydi, A., Decotignie, J.-D., and Peiris, V. 2004. WiseNET: An Ultralow-Power Wireless Sensor Network Solution. *Computer 37*, 62–70.

[35] Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. 1999. Next Century Challenges: Scalable Coordination in Sensor Networks. In *MobiCom'99: Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*. ACM, New York, NY, USA, 263–270.

[36] Fauth, A., Van Praet, J., and Freericks, M. 1995. Describing Instruction Set Processors using nML. In *EDTC'95: Proceedings of the 1995 European Conference on Design and Test*. IEEE Computer Society, Washington, DC, USA, 503.

[37] Fin, A., Fummi, F., and Perbellini, G. 2001. Soft-Cores Generation by Instruction Set Analysis. In *ISSS'01: Proceedings of the 14th International Symposium on Systems Synthesis*. ACM, 227–232.

[38] F.L. Lewis. 2005. *Wireless Sensor Networks*. Book Chapter in Smart Environments: Technologies, Protocols, Applications.

[39] Fonseca, R., Dutta, P., Levis, P., and Stoica, I. 2008. Quanto: Tracking Energy in Networked Embedded Systems. In *OSDI'08: USENIX Symposium on Operating Systems Design and Implementation*. 323–338.

[40] Forte. 2010. Cynthesizer and High-Level Design. Product.

[41] Fraser, C. W., Henry, R. R., and Proebsting, T. A. 1992. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not. 27,* 4.

[42] Gajski, D. and Reshadi, M. 2005. A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths. In *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 21 –26.

[43] Garrett, D., Stan, M., and Dean, A. 1999. Challenges in Clockgating for a Low Power ASIC Methodology. In *ISLPED'99: Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. 176–181.

[44] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not. 38,* 5, 1–11.

[45] Glokler, T., Bitterlich, S., and Meyr, H. 2000. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *SOCC'00: Proceedings of the13th Annual IEEE International ASIC/SOC Conference*. 102 –106.

[46] Gonzalez, R. 2000. Xtensa: A Configurable and Extensible Processor. *IEEE Micro 20,* 2 (mar/apr), 60 –70.

[47] Grant Martin and Gary Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design and Test of Computers 26*, 18–25.

[48] Gupta, Rajesh and Brewer, Forrest. 2008. High-Level Synthesis: A Retrospective. In *High-Level Synthesis*, Coussy, Philippe and Morawiec, Adam, Ed. Springer Netherlands, Chapter 2, 13–28.

[49] H. Ritter. 2010. ScatterWeb.

[50] Hadjiyiannis, G., Hanono, S., and Devadas, S. 1997. ISDL: An Instruction Set Description Language for Retargetability. In *DAC'97: Proceedings of the 34th Annual Design Automation Conference*. ACM, New York, NY, USA, 299–302.

[51] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. 1999. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *DATE'99: Proceedings of Design, Automation and Test in Europe Conference and Exhibition.* 485 –490.

[52] He, T., Krishnamurthy, S., Luo, L., Yan, T., Gu, L., Stoleru, R., Zhou, G., Cao, Q., Vicaire, P., Stankovic, J. A., Abdelzaher, T. F., Hui, J., and Krogh, B. 2006. VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance. *ACM Transactions on Sensor Networks 2,* 1, 1–38.

[53] Hempstead, M., Wei, G.-Y., and Brooks, D. 2009. An Accelerator-Based Wireless Sensor Network Processor in 130nm CMOS. In *CASES'09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, New York, NY, USA, 215–222.

[54] Herring, C. and Kaplan, S. 2000. Component-Based Software Systems for Smart Environments. *Personal Communications, IEEE [see also IEEE Wireless Communications] 7,* 5 (Oct), 60–61.

[55] Hill, J., Horton, M., Kling, R., and Krishnamurthy, L. 2004. The Platforms Enabling Wireless Sensor Networks. *ACM Commun. 47,* 6, 41–46.

[56] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System Architecture Directions for Networked Sensors. *SIGPLAN Not. 35,* 11, 93–104.

[57] Hill, J. L. and Culler, D. E. 2002. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro 22*, 12–24.

[58] Hoffmann, A., Schliebusch, O., Nohl, A., Braun, G., Wahlen, O., and Meyr, H. 2001. A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) using the Machine Description Language LISA. In *ICCAD'01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design.* Piscataway, NJ, USA, 625–630.

[59] Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., and Bose, P. 2004. Microarchitectural Techniques for Power Gating of Execution Units. In *ISLPED'04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design.* 32–37.

[60] Huang, I.-J. and Despain, A. 1994. Synthesis of Instruction Sets for Pipelined Microprocessors. In *DAC'94: Proceedings of the 31st ACM/IEEE Design Automation Conference.* 5–11.

[61] Hwang, C.-T., Lee, J.-H., and Hsu, Y.-C. 1991. A Formal Approach to the Scheduling Problem in High Level Synthesis . *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10,* 4 (apr.), 464 –475.

[62] INRIA. 2010a. WSim, A Simulator for Microcontroller-based Wireless Platforms. Tech. Project.

[63] INRIA. 2010b. WSNet, An Event-Driven Simulator for Large Scale Wireless Sensor Networks. Tech. Project.

[64] INRIA, Tech. Project. 2010. PowWow, Protocol for Low Power Wireless Sensor Network.

[65] Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MobiCom '00: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. ACM, New York, NY, USA, 56–67.

[66] Intel. 1999. StrongArm SA-1100, Data Sheet. Tech. Report.

[67] Ishihara, T. and Asada, K. 2001. A System Level Memory Power Optimization Technique using Multiple Supply and Threshold Voltages. In *ASP-DAC'01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, New York, NY, USA, 456–461.

[68] Jain, M., Balakrishnan, M., and Kumar, A. 2001. ASIP Design Methodologies: Survey and Issues. In *VLSI'01: Proceedings of the 14th International Conference on VLSI Design*. 76 –81.

[69] Kao, J. and Chandrakasan, A. 2000. Dual-Threshold Voltage Techniques for Low-Power Digital Circuits. *IEEE Journal of Solid-State Circuits 35,* 7 (July), 1009 –1018.

[70] Karl, H. and Willig, A. 2005. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons.

[71] Kastrup, B. 2001. Automatic Synthesis of Reconfigurable Instruction Set Accelerators. Ph.D. thesis, Eindhoven University of Technology.

[72] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. 2007. Health Monitoring of Civil Infrastructures using Wireless Sensor Networks. In *IPSN'07: Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, New York, NY, USA, 254–263.

[73] Kim Tae-Hyung, H. S. 2005. State Machine Based Operating System Architecture for Wireless Sensor Networks. In *PDCAT'05: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*. 803–806.

[74] Kin, J., Lee, C., Mangione-Smith, W. H., and Potkonjak, M. 1999. Power Efficient Mediaprocessors: Design Space Exploration. In *DAC'99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, 321–326.

[75] Kitahara, T., Minami, F., Ueda, T., Usami, K., Nishio, S., Murakata, M., and Mitsuhashi, T. 1998. A Clock-Gating Method for Low-Power LSI Design. In *ASP-DAC'98: Proceedings of the Asia and South Pacific Design Automation Conference*. 307–312.

[76] Koes, D. R. and Goldstein, S. C. 2008. Near-Optimal Instruction Selection on DAGs. In *CGO'08: Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, New York, NY, USA, 45–54.

[77] Kum, K.-I. and Sung, W. 2001. Combined Word-Length Optimization and High-Level Synthesis of Digital Signal Processing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20,* 8 (aug.), 921 –930.

[78] Kurdahi, F. and Parker, A. 1987. REAL: A Program for REgister ALlocation. In *DAC'87: Proceedings of the 24th IEEE/ACM Design Automation Conference*. 210–215.

[79] Kwong, J., Ramadass, Y., Verma, N., Koesler, M., Huber, K., Moormann, H., and Chandrakasan, A. 2008. A 65nm Sub-Vt Microcontroller with Integrated SRAM and Switched-Capacitor DC-DC Converter. In *ISSCC'08: Proceedings of the IEEE International Solid-State Circuits Conference*. 318–616.

[80] Lach, J. and Kumar, V. 2005. Highly Flexible Multi-Mode System Synthesis. In *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 27–32.

[81] Levis, P. and Culler, D. 2002. Mate: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS'02: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA, USA, 85–95.

[82] Li, Y., Wang, Z., and Song, Y. 2006. Wireless Sensor Network Design for Wildfire Monitoring. In *WCICA'06: Proceedings of the 6th World Congress on Intelligent Control and Automation*. Vol. 1. 109–113.

[83] Liao, S., Devadas, S., Keutzer, K., and Tjiang, S. 1995. Instruction Selection using Binate Covering for Code Size Optimization. In *ICCAD'95: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*. San Jose, CA, USA, 393–399.

[84] Lin, E.-Y., Rabaey, J., and Wolisz, A. 2004. Power-Efficient Rendez-Vous Schemes for Dense Wireless Sensor Networks. In *ICC'04: Proceedings of the IEEE International Conference on Communications*. Vol. 7. 3769–3776.

[85] Lin Zhong and Jha, N.K. 2005. Interconnect-Aware Low-Power High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24,* 3 (march), 336 – 351.

[86] L.L'Hours. 2005. Generating Efficient Custom FPGA Soft-Cores for Control-Dominated Applications. In *ASAP'05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors*. Washington, DC, USA, 127–133.

[87] Long, C. and He, L. 2003. Distributed Sleep Transistor Network for Power Reduction. In *DAC'03: Proceedings of the 40th annual ICM/IEEE Design Automation Conference*. ACM, 181–186.

[88] Mahnke, T., Stechele, W., and Hoeld, W. 2002. Dual Supply Voltage Scaling in a Conventional Power-Driven Logic Synthesis Environment. In *PATMOS'02: Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*. Springer-Verlag, London, UK, 146–155.

[89] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. 2002. Wireless Sensor Networks for Habitat Monitoring. In *WSNA'02: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*. ACM, New York, NY, USA, 88–97.

[90] Malan, D., Fulford-jones, T., Welsh, M., and Moulton, S. 2004. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. In *BSN'04: International Workshop on Wearable and Implantable Body Sensor Networks*.

[91] Martin, K., Wolinski, C., Kuchcinski, K., Floch, A., and Charot, F. 2009. Constraint-Driven Instructions Selection and Application Scheduling in the DURASE System. In *ASAP'09: Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society, Boston, MA, USA, 145 – 152.

[92] Mentor Graphics. 2010. Catapult C Synthesis: Full-Chip High-Level Synthesis. Product.

[93] Messé, V. 1999. Production de Compilateurs Flexibles pour la Conception de Processeurs Programmables Spécialisés. Ph.D. thesis, Université de Rennes-1.

[94] Moreano, N., Borin, E., de Souza, C., and Araujo, G. 2005. Efficient Datapath Merging for Partially Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24,* 7 (jul.), 969 – 980.

[95] Mutoh, S., Douseki, T., Matsuya, Y., Aoki, T., Shigematsu, S., and Yamada, J. 1995. 1-V Power Supply High-Speed Digital Circuit Technology with Multithreshold-Voltage CMOS. *IEEE Journal of Solid-State Circuits 30,* 8 (aug), 847 –854.

[96] Mysore, S., Agrawal, B., Chong, F., and Sherwood, T. 2008. Exploring the Processor and ISA Design for Wireless Sensor Network Applications. In *VLSI'08: Proceedings of the 21st International Conference on VLSI Design*. 59–64.

[97] Nazhandali, L., Minuth, M., and Austin, T. 2005. SenseBench: Toward an Accurate Evaluation of Sensor Network Processors. In *IISWC'05: Proceedings of the IEEE International Workload Characterization Symposium*. Austin, Texas, USA, 197–203.

[98] NEC. 2010. CyberWorkBench: Pioneering C-based LSI Design. Product.

[99] Nguyen, T.-D., Berder, O., and Sentieys, O. 2007. Cooperative MIMO Schemes Optimal Selection for Wireless Sensor Networks. In *VTC2007-Spring: Proceedings of the 65th IEEE Vehicular Technology Conference*. Dublin, Ireland, 85–89.

[100] NXP Semiconductors. 2010. LPC1111/12/13/14, 32-bit ARM Cortex-M0 microcontroller. Product data sheet.

[101] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler. 2005. TinyOS: An Operating System for Sensor Networks . In *Ambient Intelligence*. Springer.

[102] Pangrle, B. 1988. Splicer: A Heuristic Approach to Connectivity Binding. In *DAC'88: Proceedings of the 25th annual IEEE/ACM Design Automation Conference*. 536–541.

[103] Pasha, M. A., Derrien, S., and Sentieys, O. 2009. Ultra Low-Power FSM for Control Oriented Applications. In *ISCAS'09: IEEE International Symposium on Circuits and Systems*. Taipei, Taiwan, 1577–1580.

[104] Pasha, M. A., Derrien, S., and Sentieys, O. 2010a. A Complete Design-Flow for the Generation of Ultra Low-Power WSN Node Architectures Based on Micro-Tasking. In *DAC'10: Proceedings of the 47th ACM/IEEE Design Automation Conference*. ACM, Anaheim, CA, USA, 693–698.

[105] Pasha, M. A., Derrien, S., and Sentieys, O. 2010b. System Level Synthesis for Ultra Low-Power Wireless Sensor Nodes. In *DSD'10: Proceedings of the 13th Euromicro Conference on Digital System Design*. Lille, France, 493–500.

[106] Paulin, P. and Knight, J. 1989. Algorithms for High-Level Synthesis. *IEEE Design and Test of Computers 6,* 6 (dec.), 18 –31.

[107] Paulin, P., Knight, J., and Girczyc, E. 1986. HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis. In *DAC'86: Proceedings of the 23rd annual IEEE/ACM Design Automation Conference*. 263–270.

[108] Pedram, M. 1996. Power Minimization in IC Design: Principles and Applications. *ACM Transaction on Design Automation of Electronic Systems 1,* 1, 3–56.

[109] Piguet, C., Masgonty, J.-M., Arm, C., Durand, S., Schneider, T., Rampogna, F., Scarnera, C., Iseli, C., Bardyn, J.-P., Pache, R., and Dijkstra, E. 1997. Low-Power Design of 8-b Embedded CoolRisc Microcontroller Cores. *IEEE Journal of Solid-State Circuits 32,* 7 (jul), 1067 –1078.

[110] Polastre, J., Hill, J., and Culler, D. 2004. Versatile Low Power Media Access for Wireless Sensor Networks. In *SenSys'04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 95–107.

[111] Polastre, J., Szewczyk, R., and Culler, D. 2005. Telos: Enabling Ultra-Low Power Wireless Research. In *IPSN'05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. IEEE Press, Piscataway, NJ, USA, 364–369.

[112] Pottie, G. J. and Kaiser, W. J. 2000. Wireless Integrated Network Sensors. *ACM Commun. 43,* 5, 51–58.

[113] Pradhan, S. N., Kumar, M. T., and Chattopadhyay, S. 2008. Integrated Power-Gating and State Assignment for Low Power FSM Synthesis. In *IVLSI'08: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. Vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, 269–274.

[114] Proebsting, T. A. 1995. BURS Automata Generation. *ACM Transactions on Programming Languages and Systems 17,* 3, 461–486.

[115] Rabaey, J. M., Ammer, M. J., da Silva, J. L., Patel, D., and Roundy, S. 2000. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. *Computer 33,* 7, 42–48.

[116] Raghunathan, V., Schurgers, C., Park, S., and Srivastava, M. 2002. Energy-aware Wireless Microsensor Networks. *IEEE Signal Processing Magazine 19,* 2 (Mar).

[117] Raghunathan, A. and Jha, N.K. 1997. SCALP: An Iterative-Improvement-Based Low-Power Data Path Synthesis System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16,* 11 (nov), 1260 –1277.

[118] Rahman, A., Das, S., Tuan, T., and Trimberger, S. 2006. Determination of Power Gating Granularity for FPGA Fabric. In *CICC'06: Proceedings of the IEEE Custom Integrated Circuits Conference*. 9–12.

[119] Raval, R. K., Fernandez, C. H., and Bleakley, C. J. 2010. Low-Power TinyOS Tuned Processor Platform for Wireless Sensor Network Motes. *ACM Transactions on Design Automation of Electronic Systems 15,* 3, 1–17.

[120] RFM. 2008. 916.50 MHz Hybrid Transceiver.

[121] Schurgers, C. and Srivastava, M. 2001. Energy Efficient Routing in Wireless Sensor Networks. In *MILCOM'01: Proceedings of the Military Communications Conference*. Vol. 1. 357–361.

[122] Seok, M., Hanson, S., Lin, Y.-S., Foo, Z., Kim, D., Lee, Y., Liu, N., Sylvester, D., and Blaauw, D. 2008. The Phoenix Processor: A 30pW Platform for Sensor Applications. In *VLSI'08: Proceedings of the IEEE Symposium on VLSI Circuits*. 188–189.

[123] Sheets, M., Burghardt, F., Karalar, T., Ammer, J., Chee, Y., and Rabaey, J. 2006. A Power-Managed Protocol Processor for Wireless Sensor Networks. In *VLSI'06: Proceedings of the IEEE Symposium on VLSI Circuits*. 212–213.

[124] Silicon Hive. 2010. Exploiting Parallelism, while Managing Complexity using Silicon Hive Programming Tools. Product.

[125] Smith, J. 2009. Rapid Implementation of Low Power Microprocessors. *DACeZine 4,* 2.

[126] Synposys. 2010. High Level Synthesis with Synphony C Compiler. Product.

[127] Tan, C. H. and Allen, J. 1994. Minimization of Power in VLSI Circuits Using Transistor Sizing, Input Ordering, and Statistical Power Estimation. In *Proceedings of the International Workshop on Low Power Design*. 75–80.

[128] Tensilica. 2010. Tensilica: Customizable Processor Cores for the Dataplane. Product.

[129] Texas Instruments. 2009. MSP430 User Guide. Tech. Report.

[130] Texas Instruments. 2010a. MSP430 16-bit Ultra-Low Power MCUs.

[131] Texas Instruments. 2010b. Single-Chip 2.4 GHz IEEE 802.15.4 Compliant and ZigBee RF Transceiver.

[132] Texas Instruments. 2010c. Single Chip Ultra Low Power RF Transceiver for 315/433/868/915 MHz SRD Band.

[133] The Eclipse Foundation. 2010a. Eclipse Development Platform.

[134] The Eclipse Foundation. 2010b. Eclipse Modeling Framework (EMF).

[135] The Eclipse Foundation. 2010c. Java Emitter Template (JET).

[136] The Eclipse Foundation. 2010d. Xtext, a Framework for Development of Textual Domain Specific Languages (DSLs).

[137] Tseng, C.-J. and Siewiorek, D. 1986. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 5,* 3 (jul.), 379 – 395.

[138] University of California, Berkeley. 1999. Tech. Project: Smart Dust.

[139] Usami, K. and Horowitz, M. 1995. Clustered Voltage Scaling Technique for Low-Power Design. In *ISLPED'95: Proceedings of the 1995 International Symposium on Low Power Design*. ACM, New York, NY, USA, 3–8.

[140] Van der Werf, A., Peek, M. J. H., Aarts, E. H. L., van Meerbergen, J. L., Lippens, P. E. R., and Verhaegh, W. F. J. 1992. Area Optimization of Multi-Functional Processing Units. In *ICCAD'92: Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, Los Alamitos, CA, USA, 292–299.

[141] Virage Logic. 2010. ARC 700 Core Family Power-Efficient, High Performance 32-Bit Configurable CPU Cores. Product.

[142] Werner-Allen, G., Johnson, J., Ruiz, M., Lees, J., and Welsh, M. 2005. Monitoring Volcanic Eruptions with A Wireless Sensor Network. In *EWSN'05: Proceedings of the 2nd European Workshop on Wireless Sensor Networks*. 108–120.

[143] Wu, Q., Pedram, M., and Wu, X. 2000. Clock-Gating and its Application to Low Power Design of Sequential Circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications 47,* 3 (Mar), 415–420.

[144] Xilinx. 2010. MicroBlaze Soft Processor Core. Product.

[145] Ye, W., Heidemann, J., and Estrin, D. 2002. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *INFOCOM'02: Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3. 1567–1576.

[146] Zhou, Hai-Ying, and Hou, K.-M. 2007. LIMOS: A Lightweight Multi-Threading Operating System dedicated to Wireless Sensor Networks. In *WiCom'07: Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing*. 3051–3054.

List of Figures

List of Tables

Résumé :

La conception d'une plate-forme matérielle pour un nœud de réseaux de capteurs (RdC) est un véritable défi car elle est soumise à des contraintes sévères. La consommation d'énergie est souvent considérée comme la contrainte la plus forte donnée la petite taille et les besoins d'autonomie d'un nœud. De nos jours, les nœuds s'appuient sur des microcontrôleurs (*MCUs*) faible consommation disponibles dans le commerce. Ces *MCUs* ne sont pas adaptés au contexte de RdC car ils sont basés sur une structure de calcul généraliste et ils consomment trop d'énergie par rapport au budget d'énergie d'un nœud. Dans cette thèse, nous proposons un flot de conception complet, depuis le niveau système, se basant sur le concept de micro-tâches matérielles avec coupure de la tension d'alimentation (*Power Gating*). Dans cette approche, l'architecture d'un nœud est constituée d'un ensemble de micro-tâches matérielles qui sont activées selon un principe événementiel, chacune étant dédiée à une tâche spécifique du système (ex. la couche MAC, le routage, etc.). Ces micro-tâches sont gérées par un ordonnanceur matériel (*System Monitor*) qui est automatiquement généré à partir d'une description système, dans un langage spécifique (*DSL)*, du graphe des tâches d'un nœud de RdC. En combinant la spécialisation du matériel et la technique du *power gating*, nous réduisons considérablement les énergies dynamique et statique d'un nœud de RdC. Les résultats montrent que des gains en énergie dynamique de 1 à 2 ordres de grandeur sont possibles par rapport aux mises en œuvre à base des *MCUs* (ex. le MSP430). De plus, des gains de 1 ordre de grandeur en énergie statique sont également obtenus grâce à l'utilisation du *power gating*.

Abstract:

Wireless Sensor Networks (WSN) is a new and challenging research field for embedded system design automation. Engineering a WSN node platform is a tough challenge, as the design must enforce many severe constraints among which energy consumption is often the most critical one due to the small size of a node and its difficult access after deployment. WSN nodes have until now been designed using commercial low-power microcontrollers (MCUs). These MCUs are not well-suited for WSN node design as they are based on a general purpose compute engine and consume too much power w.r.t. WSN node's power budget. In this thesis, we propose a complete system-level design-flow for an alternative approach based on the concept of power-gated hardware micro-tasks. In this approach, WSN node architecture is made of several micro-tasks that are activated on an event-driven basis, each of them being dedicated to a specific task of the system (such as event-sensing, MAC, routing, etc.). These hardware micro-tasks are controlled by a hardware scheduler (called the System Monitor) that is automatically generated from a system-level description of the WSN node task graph in the form of a textual Domain Specific Language (DSL). By combining hardware specialization with power-gating, we can drastically reduce both dynamic and static energy of a WSN node controller. The results show that dynamic power savings by one to two orders of magnitude are possible w.r.t. the software implementations based on MCUs such as the MSP430. Similarly, static power savings of one order of magnitude are also obtained due to the reduction in data memory size and power-gating.